O'REILLY®

TURING 图灵程序设计丛书

Unity移动游戏开发

Mobile Game Development with Unity

针对游戏开发零经验读者

通过完整游戏开发实例快速上手

[澳] 乔恩·曼宁　帕里斯·巴特菲尔德-艾迪生 著

赵利通 译

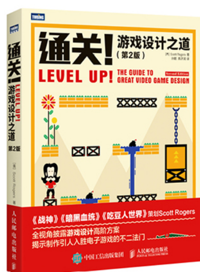


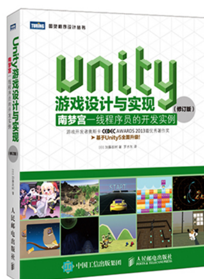
中国工信出版集团

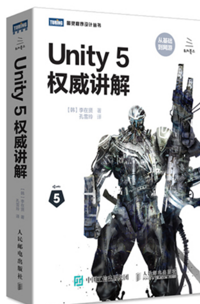


人民邮电出版社
POSTS & TELECOM PRESS

相关图书


通关！游戏设计之道
（第2版）
LEVEL UP!
THE GUIDE TO GREAT VIDEO GAME DESIGN
《战神》《暗黑血统》《吃豆人世界》策划Scott Rogers
全视角披露游戏设计高阶方案
揭示制作引人入胜电子游戏的不二法门


Unity
游戏设计与实现
（修订版）
南梦宫一线程序员的开发实例
基于Unity5全面升级!


Unity 5
权威讲解

数字版权声明

图灵社区的电子书没有采用专有客户端，您可以在任意设备上，用自己喜欢的浏览器和PDF阅读器进行阅读。

但您购买的电子书仅供您个人使用，未经授权，不得进行传播。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。



图 4-1：最终的游戏效果

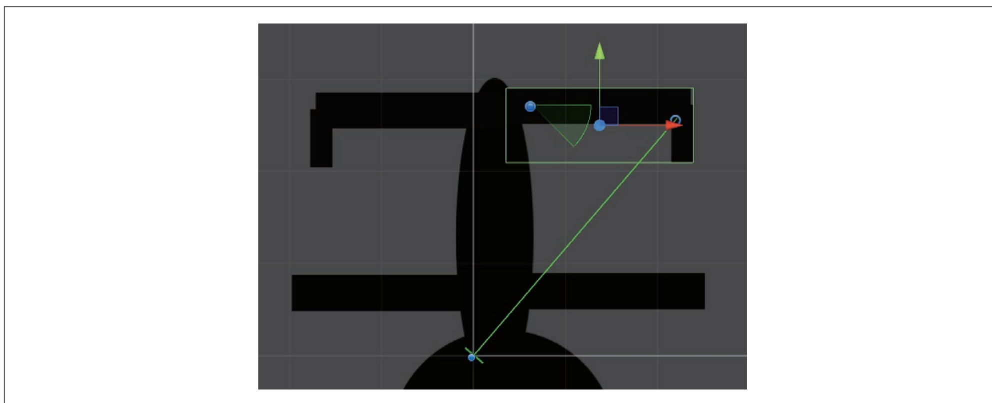


图 4-14：添加精灵关节，以将腿部连接到绳索；关节的 Anchor 靠近脚趾

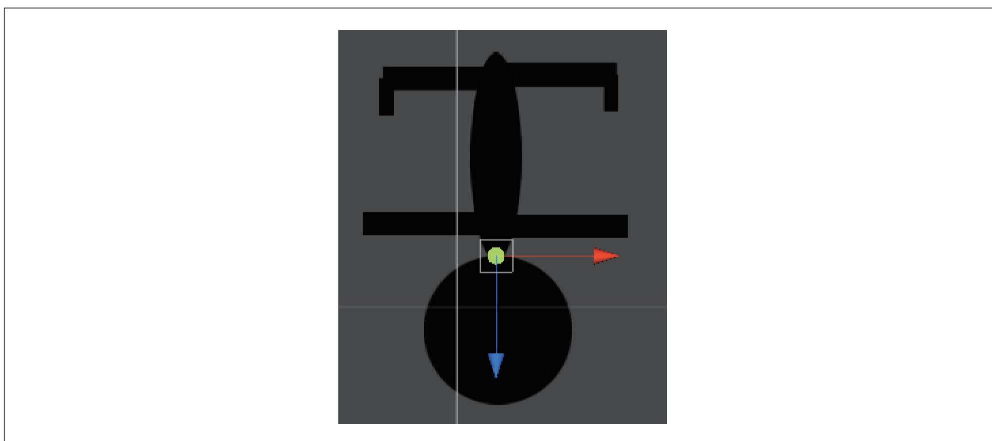


图 5-12：Head（头部）血流的位置和旋转

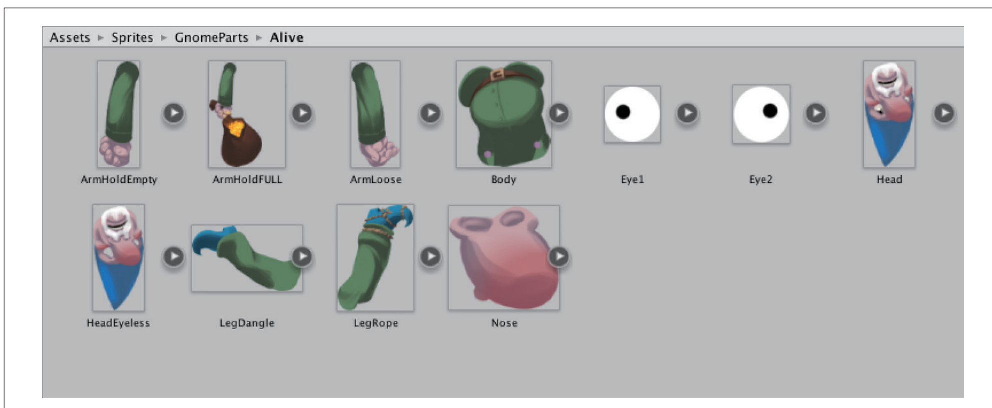


图 7-2：地精的生存精灵

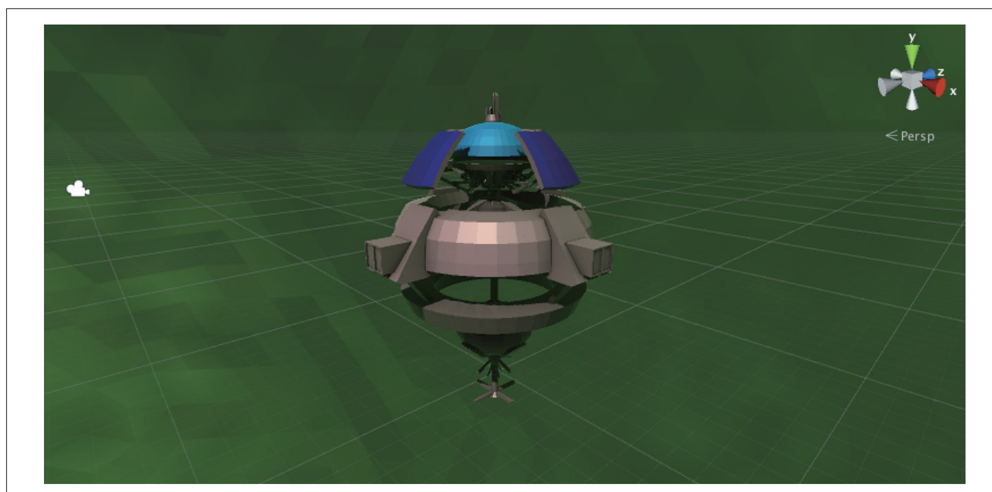


图 9-17：使用了天空盒



图 11-5：Fire 按钮

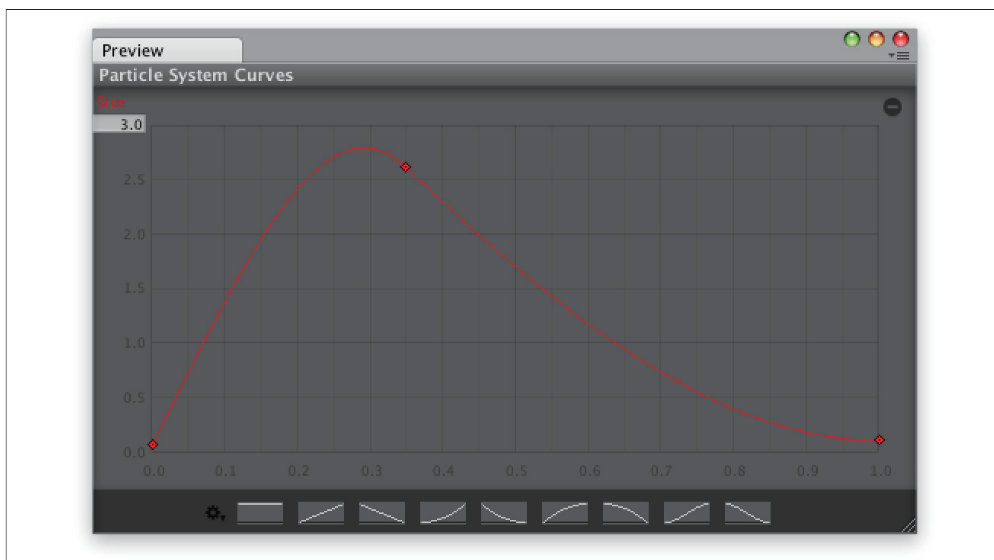


图 12-9：爆炸火球的 Size over Lifetime 曲线

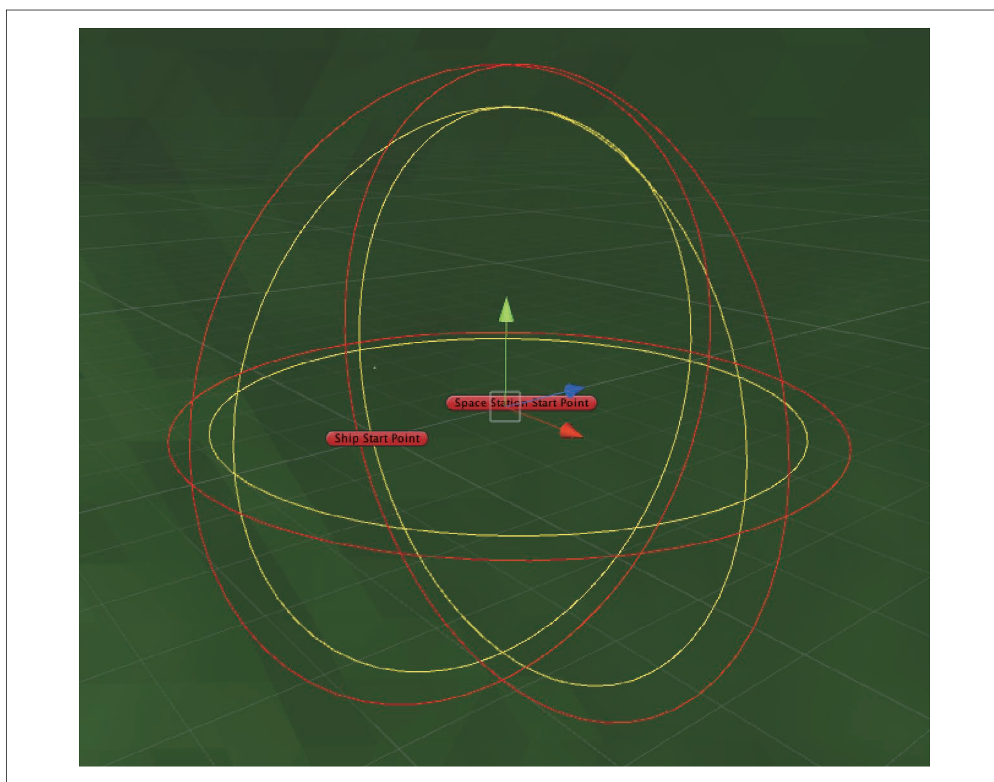


图 13-8：边界



图 13-14：正在进行中的游戏

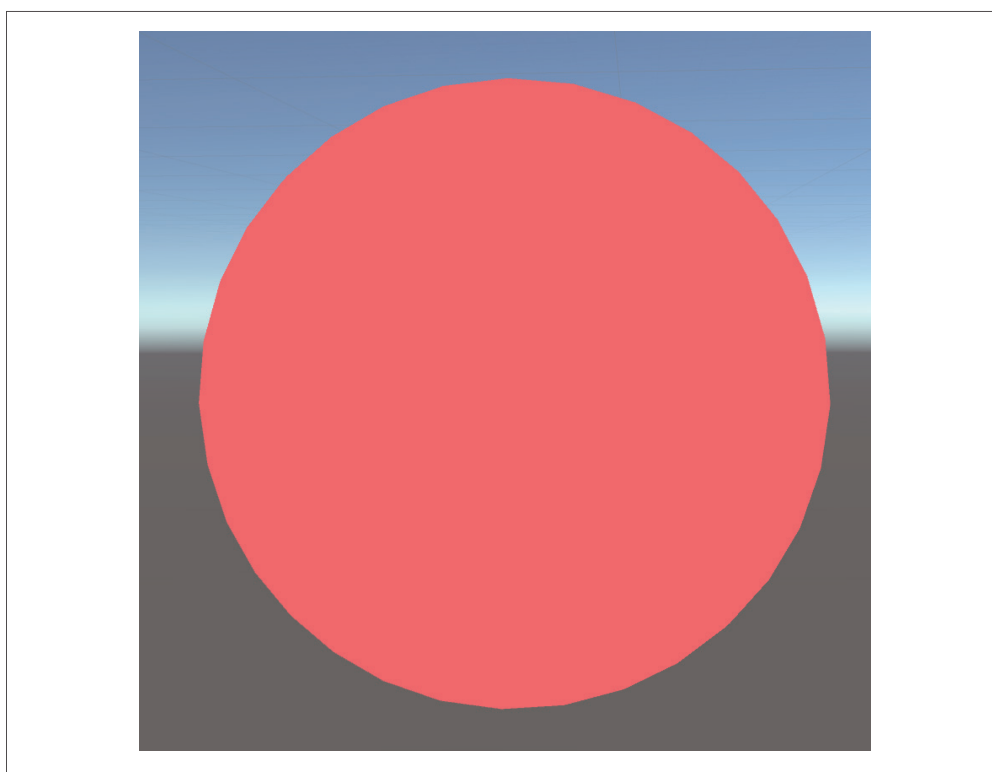


图 14-6：使用亚光色渲染后的球体

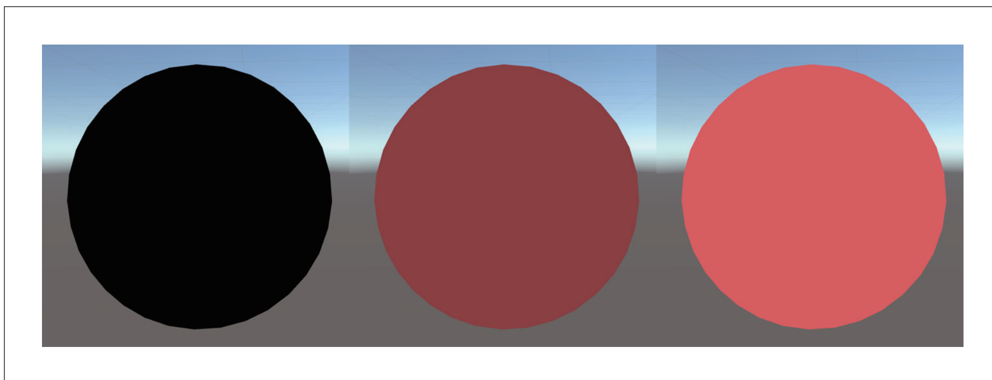


图 14-7：对象淡入并淡出

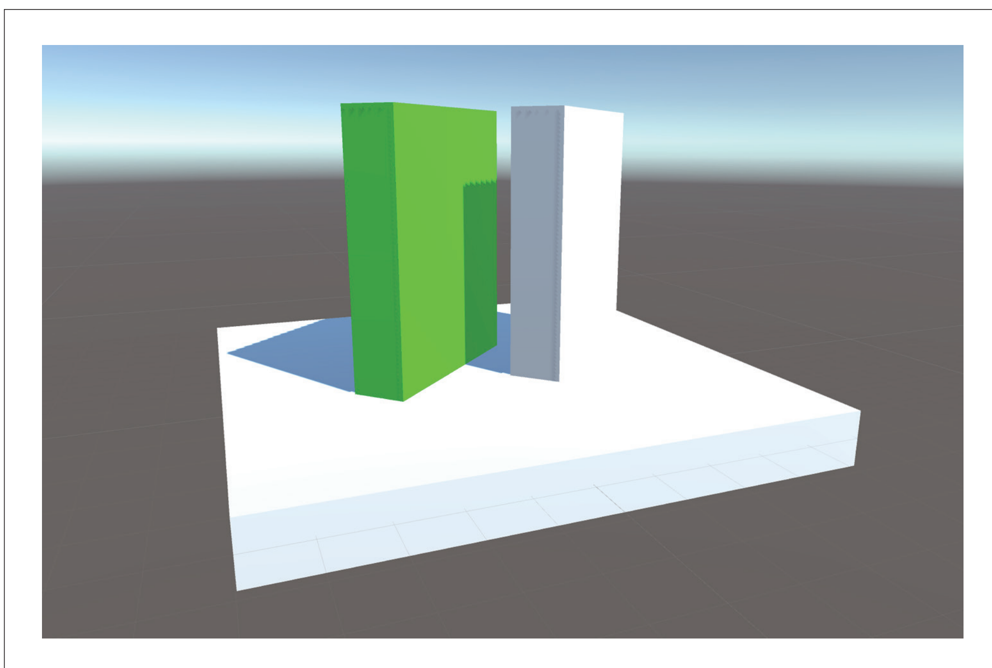


图 14-9：未使用光照贴图的场景

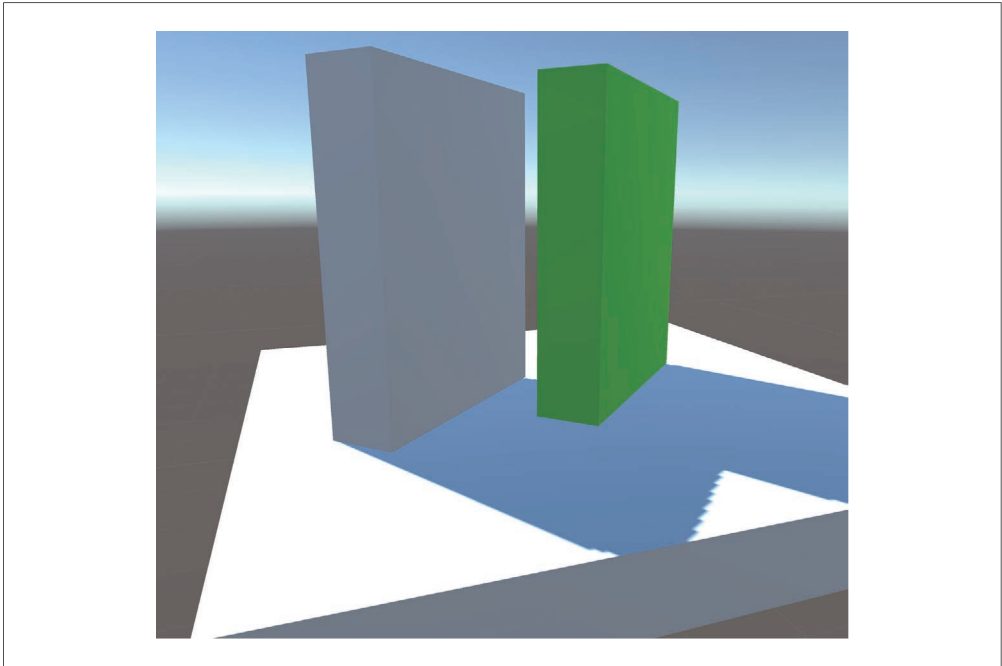


图 14-11：未激活全局光照的场景

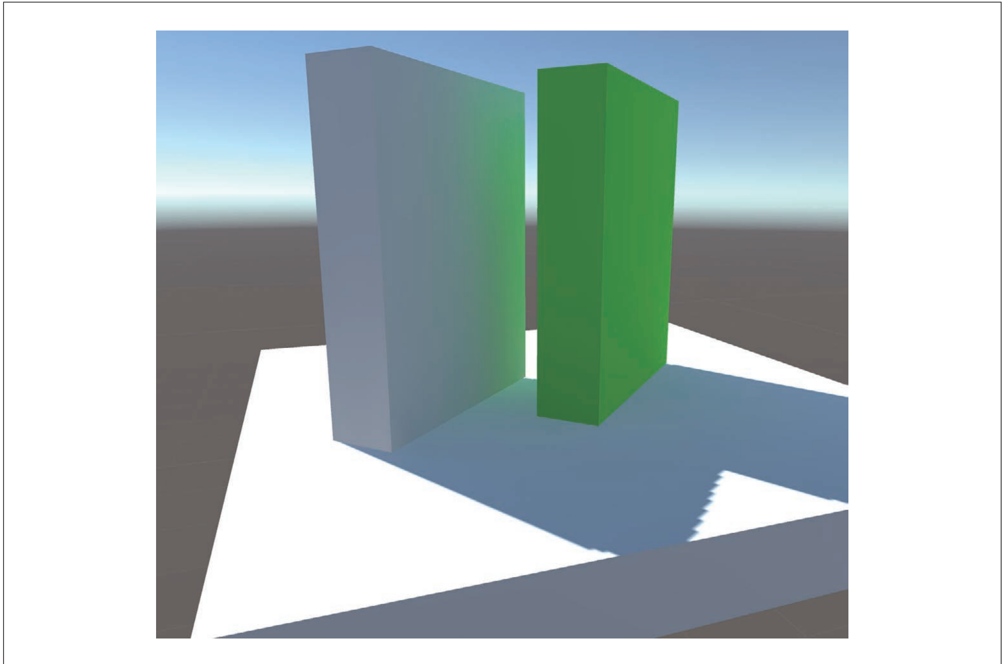


图 14-12：激活全局光照的场景，注意白色墙壁上的绿色反光

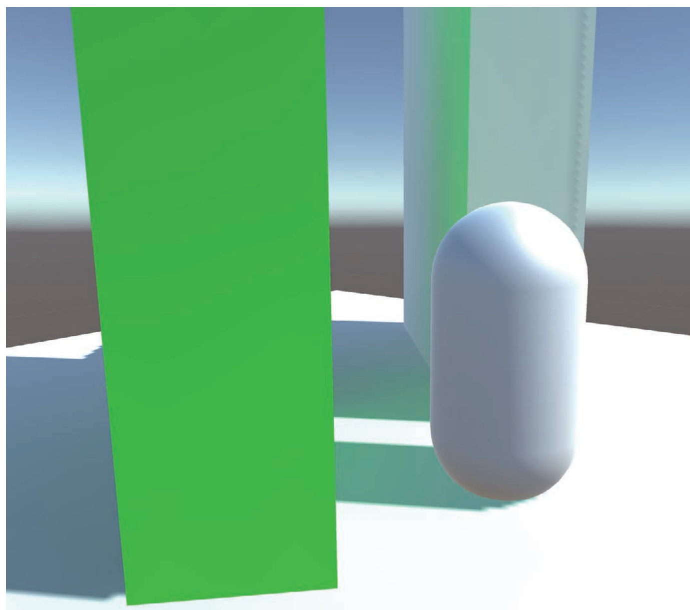


图 14-13：未使用灯光探测器的场景

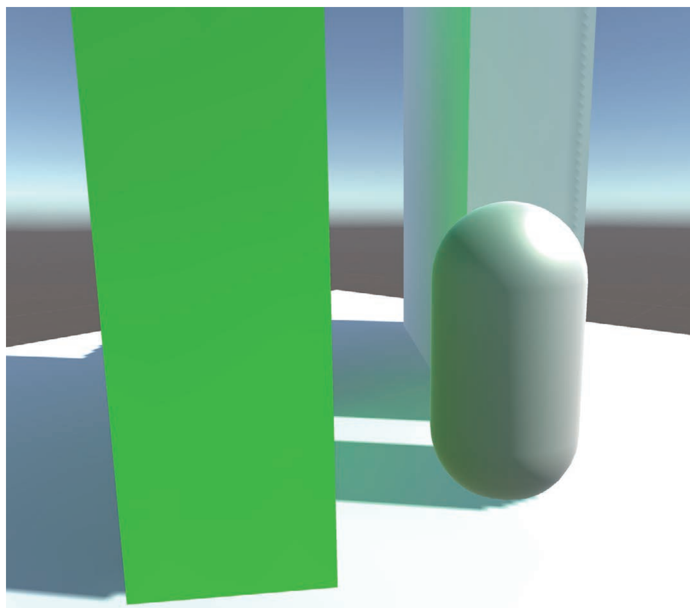


图 14-14：使用了灯光探测器的场景，注意绿色光线反射到了胶囊体上



图灵程序设计丛书

Unity移动游戏开发

Mobile Game Development with Unity

Build Once, Deploy Anywhere

[澳] 乔恩·曼宁　帕里斯·巴特菲尔德–艾迪生　著
赵利通　译

Beijing • Boston • Farnham • Sebastopol • Tokyo　O'REILLY®

O'Reilly Media, Inc.授权人民邮电出版社出版

人 民 邮 电 出 版 社
北　　京

图书在版编目（CIP）数据

Unity移动游戏开发 / （澳）乔恩·曼宁，（澳）帕里斯·巴特菲尔德－艾迪生著；赵利通译. -- 北京：人民邮电出版社，2018.8
（图灵程序设计丛书）
ISBN 978-7-115-48879-4

Ⅰ. ①U… Ⅱ. ①乔… ②帕… ③赵… Ⅲ. ①游戏程序一程序设计 Ⅳ. ①TP317.6

中国版本图书馆CIP数据核字(2018)第164579号

内 容 提 要

本书从自上而下的角度介绍了 Unity 游戏引擎的功能，并提供了具体的、面向项目的指导，说明了如何在真实的游戏场景中使用这些功能，以及如何从头开始构建让玩家爱不释手的 2D 和 3D 游戏。主要内容有：探索 Unity 的基础功能；构建卷轴动作游戏；创建具有炮弹射击和重生对象功能的 3D 空战模拟游戏；深入了解 Unity 的高级功能。

本书适合游戏开发人员阅读。

◆ 著　　[澳] 乔恩·曼宁　帕里斯·巴特菲尔德-艾迪生
译　　赵利通
责任编辑　温　雪
责任印制　周昇亮

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　印刷

◆ 开本：800×1000　1/16
印张：19.75　　彩插：4
字数：466千字　　2018年 8 月第 1 版
印数：1－3 500册　　2018年 8 月北京第 1 次印刷

著作权合同登记号　图字：01-2018-3479号

定价：89.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京东工商广登字 20170147 号

版权声明

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 *Make* 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过图书出版、在线服务或者面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

目录

第三部分　构建一个 3D 游戏：太空射击游戏

第四部分 高级功能

前言

欢迎阅读本书！我们将带领你从零开始开发两个完整的游戏，并在此过程中教会你基础的和高级的 Unity 概念及技术。

本书分为 4 个部分。

第一部分介绍 Unity 游戏引擎，并探索引擎的基础内容，包括如何安排游戏、图形、脚本、音效、物理和粒子系统的结构。第二部分带领你使用 Unity 创建一个完整的 2D 游戏，内容是一个悬在绳子上的地精试图获得宝藏。第三部分探索如何使用 Unity 创建一个完整的 3D 游戏，包括飞船、小行星等。第四部分探索 Unity 的一些更高级的功能，包括光照、GUI 系统、扩展 Unity 编辑器、Unity 资源商店、部署游戏，以及平台特定的功能。

如果你有任何反馈，可发送邮件到 unitybook@secretlab.com.au。

资源下载

本书补充资料（美术作品、音效、代码示例、练习题、勘误等）的下载地址为：https://secretlab.com.au/books/unity。[1]

目标读者和阅读方法

本书适合想要开发游戏，但是没有任何游戏开发经验的读者。

Unity 支持几种不同的编程语言，本书使用 C#。本书假定你知道如何使用一种现代语言编写程序，但只要熟悉编程的基本内容即可，无须在近期编写过程序。

Unity 编辑器能够在 macOS 和 Windows 上运行。我们使用的是 macOS，所以本书中给出的屏幕截图都是 macOS 上的截图，但是书中的内容同样适用于 Windows，仅有一点例外：使用 Unity 开发 iOS 游戏。介绍相关内容的时候，我们再进行解释，不过简单来说，在

注 1：读者也可前往本书图灵社区页面（http://www.ituring.com.cn/book/2117）获取资料并提交中文版勘误。
——编者注

Windows 上是无法开发 iOS 游戏的。Windows 上可以开发 Android 游戏，在 macOS 上则可以为 iOS 和 Android 开发游戏。

本书的内容组织方法是，在着手开发游戏之前，先介绍游戏设计以及 Unity 自身的一些基础知识。因此，本书第一部分会先介绍基础知识。之后，第二部分和第三部分将分别探索如何开发 2D 游戏和 3D 游戏。第四部分将介绍你应该知道的其他 Unity 功能。

本书假定你能够相当自如地使用你的操作系统和移动设备（iOS 或 Android）。

本书不会介绍如何创建游戏中使用的美术作品或音效资源，不过，我们为本书中开发的两个游戏提供了相关资源。

排版约定

本书使用了下列排版约定。

- **黑体**
  表示新术语或重点强调的内容。
- 等宽字体（`constant width`）
  表示程序片段，以及正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键字等。
- 加粗等宽字体（**`constant width bold`**）
  表示应该由用户输入的命令或其他文本。
- 等宽斜体（*`constant width italic`*）
  表示应该由用户输入的值或根据上下文确定的值替换的文本。



该图标表示提示或建议。



该图标表示一般注记。



该图标表示警告或警示。

使用代码示例

补充材料（代码示例、练习等）可以从 https://secretlab.com.au/books/unity 下载。

本书是要帮你完成工作的。一般来说，如果本书提供了示例代码，你可以把它用在你的程序或文档中。除非你使用了很大一部分代码，否则无须联系我们获得许可。比如，用本书的几个代码片段写一个程序就无须获得许可，销售或分发 O'Reilly 图书的示例光盘则需要获得许可；引用本书中的示例代码回答问题无须获得许可，将书中大量的代码放到你的产品文档中则需要获得许可。

我们很希望但并不强制要求你在引用本书内容时加上引用说明。引用说明一般包括书名、作者、出版社和 ISBN。比如："*Mobile Game Development with Unity* by Jon Manning and Paris Buttfield-Addison (O'Reilly). Copyright 2017 Jon Manning and Paris Buttfield-Addison, 978-1-491-94474-5。"

如果你觉得自己对示例代码的用法超出了上述许可的范围，欢迎你通过 permissions@oreilly.com 与我们联系。

O'Reilly Safari



Safari（前身为 Safari Books Online）是为企业、政府、教育机构和个人提供的会员制培训和参考平台。

会员可以访问来自 250 多家出版商的上千种图书、培训视频、学习路径、互动教程和精选播放列表。这些出版商包括 O'Reilly Media、Harvard Business Review、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Adobe、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology，等等。

欲知更多信息，请访问 https://www.safaribooksonline.com/。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书的网站地址是：http://shop.oreilly.com/product/0636920032359.do。

对于本书的评论和技术性问题，请发送电子邮件到：bookquestions@oreilly.com。

要了解更多 O’Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：http://www.oreilly.com。

我们在 Facebook 的地址如下：http://facebook.com/oreilly。

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia。

我们的 YouTube 视频地址如下：http://www.youtube.com/oreillymedia。

致谢

乔恩和帕里斯感谢本书优秀的编辑们，特别是 Brian MacDonald 和 Rachel Roumeliotis 的出色工作，让本书得以出版。感谢你们的热忱！还要感谢 O’Reilly Media 出色的员工，让写书的过程如此愉快！

感谢我们的家人支持我们的游戏开发工作，以及 MacLab 和 OSCON（你们知道我们在说谁）的鼓励和热忱。还要特别感谢我们的技术审校者，Tim Nugent 博士。

电子版

扫描如下二维码，即可购买本书电子版。



第一部分

Unity基础

本书介绍如何使用 Unity 游戏引擎有效地开发移动游戏。本部分包含 3 章，介绍了 Unity 的基础知识，帮助你从整体上了解这个应用程序，并讨论了如何使用 C# 编程语言在 Unity 中编程。

第 1 章

Unity简介

在正式探索 Unity 游戏引擎之前，我们先来介绍一些基础知识：Unity 是什么，有什么用，以及如何获取 Unity。同时，对于本书探讨的主题，我们会做一些限制，毕竟本书要讨论的是移动开发，而不是**所有类型**的开发。如果是纸质书，那样的一本书会厚得多；如果是电子书，其内容会多到让你的阅读软件崩溃。我们不会让这样不幸的事件发生在你身上。

1.1　内容简介

在开始讲解 Unity 之前，我们先来了解本书的主题：移动游戏开发。

移动游戏

什么是移动游戏？它与其他类型的游戏有什么区别？更实际的问题是，在设计和实现游戏时，这些区别会对你的决策产生什么样的影响？

15 年前的移动游戏可大致分成以下两类：

- 极其简单的游戏，交互少，画面简单，复杂度低；
- 很复杂的游戏，只在专门的移动游戏主机上提供，由能够获取昂贵的移动游戏主机开发工具包的公司开发出来。

硬件复杂度和游戏的发行者造成了这种区分。如果想开发复杂的游戏（**复杂**体现为屏幕上同时有多个物体移动），就要用到只有昂贵的便携式游戏主机（例如任天堂的手持设备）才能提供的更强大的计算能力。因为主机厂商也掌握着游戏的发行渠道，并且想拥有极大的控制权，所以获得为更强大的硬件开发游戏的许可成为了一项挑战。

不过，随着功能强大的硬件变得越来越便宜，开发人员拥有的选项也更多了。2008 年，

Apple 公司向软件开发人员提供了 iPhone 的开发工具包；同一年，Google 的 Android 平台也对软件开发人员开放。这些年来，iOS 和 Android 成为了功能极其强大的平台，移动游戏也成为了全世界最受欢迎的电子游戏。

如今的移动游戏可基本分为 3 类：

- 简单的游戏，具备精心设计的交互方式和画面，以及仔细控制的复杂度，这些是游戏设计的基石；
- 更加复杂的游戏，发布在多种平台上，包括专门的移动游戏主机和智能手机；
- 主机或 PC 游戏的移动版本。

这 3 类游戏都可用 Unity 实现，但是本书主要关注第一类。在探讨 Unity 及其用法后，我们将一步步创建具备上述特征的两个游戏。

1.2 Unity概述

在说明了我们将开发什么以后，下面介绍**开发工具**：Unity 游戏引擎。

1.2.1 Unity能够做什么

多年以来，Unity 的关注点一直是**让游戏开发平民化**，即让任何人都能开发游戏，同时让游戏能够出现在尽可能多的地方。然而，没有哪个软件开发包适合所有情形。知道 Unity 最适合哪些场景，以及在什么情况下应该考虑使用其他软件开发包，是很有帮助的。

Unity 特别适合以下场景。

为多种设备开发游戏

Unity 的多平台支持可能是业界最好的。如果想开发在多种平台（甚至只是多种**移动**平台）上运行的游戏，Unity 可能是最好的选择。

当开发速度很重要时

你当然可以用几个月的时间来开发一个包含所需功能的游戏引擎，但也完全可以使用第三方引擎，如 Unity。公平地说，也存在其他引擎，例如 Unreal 或 Cocos2D，不过这引出了我们要说明的下一点。

需要完善的功能集，但是不想开发自己的工具时

Unity 包含各种十分适合移动游戏的功能，并让开发人员能够轻松创建简单易用的内容。

尽管 Unity 具有上述优点，但是在一些场景中却不是最合适的工具，包括以下两种场景。

开发不应该频繁重绘的游戏时

Unity 不太适合一些不需要频繁绘制大量图形的游戏，因为 Unity 引擎每一帧都会重绘屏幕。对于实时动画而言，必须这么做，但是使用的资源也更多。

需要精准控制引擎行为时

除非你购买了 Unity 的源代码许可（可以购买，但是很少有人这么做），否则没有任何方法能够控制引擎最底层的行为。这并不意味着不能精细地控制 Unity（实际上在大部分情况下没必要这么做），只是说有些行为不受你的控制而已。

1.2.2 获取Unity

Unity 可在 Windows、macOS 和 Linux 上使用，分为 3 种主要版本：**个人版**、**个人加强版**和**专业版**。



在本书英文版出版时（2017 年中），Unity 对 Linux 的支持还是实验性的。

- 个人版是为独立开发人员设计的，他们可以使用 Unity 来开发自己的游戏。个人版是免费的。
- 个人加强版是为独立开发人员和小型开发团队设计的。在撰写本书时，个人加强版的价格为每月 35 美元。
- 专业版是为小型和大型开发团队设计的。在撰写本书时，专业版的价格为每月 125 美元。



Unity 也为大型开发团队设计了企业版，但是本书作者并不经常使用这个版本。

Unity 在不同版本中的功能大致相同。免费版和付费版的主要区别是，免费版会在游戏中强加一个显示 Unity 徽标的闪屏。免费版只提供给年收入不超过 10 万美元的个人或组织，而个人加强版的限制则是 20 万美元。个人加强版和专业版提供了更好的服务，例如在 Unity 的云构建服务中优先排队（17.1.2 节会更详细地讨论）。

要下载 Unity，可访问 https://store.unity.com。安装之后，就可以使用了。下一章见！

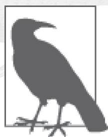
第 2 章

Unity一览

安装了 Unity 之后，花些时间熟悉其环境很有帮助。Unity 的用户界面相当直观，但是也有不少地方需要一些时间来了解。

2.1　编辑器

初次启动时，Unity 要求提供许可密钥，并登录自己的账户。如果没有账户，或者不想登录，可以跳过登录界面。



如果不登录，将无法使用云构建服务和其他服务。第 17 章将介绍 Unity 的服务。我们一开始并不会大量用到这些服务，但是登录进去还是很方便的。

完成上述步骤后，将进入 Unity 的启动界面，在这里可以选择创建一个新项目，或者打开已有的项目（如图 2-1 所示）。

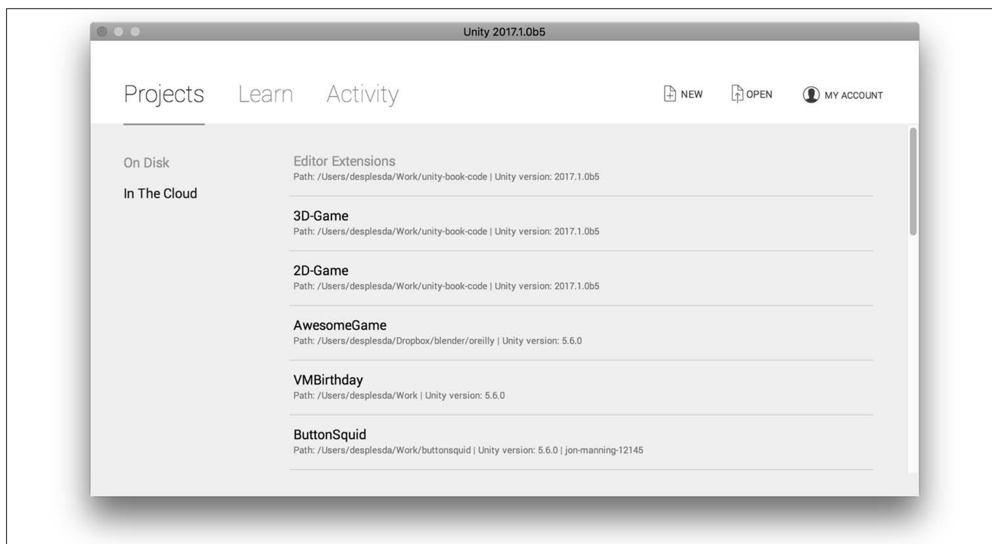


图 2-1：登录 Unity 后显示的闪屏

如果单击右上角的 New 按钮，Unity 将要求提供一些用于设置项目的信息（如图 2-2 所示），包括项目名称、保存位置，以及创建 2D 或 3D 项目。

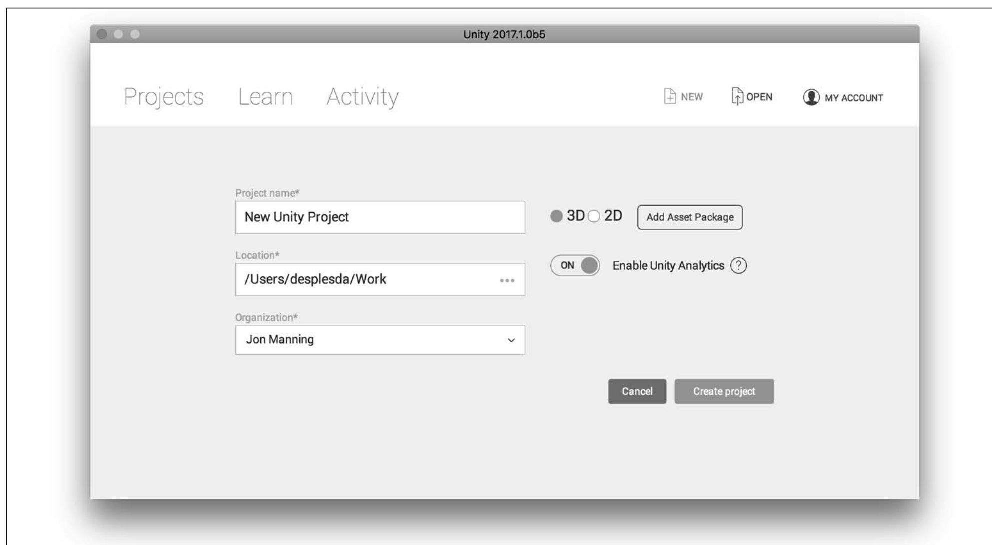


图 2-2：创建新项目



选择 2D 或者 3D 选项并不会产生巨大影响。2D 项目默认使用侧向视图，而 3D 项目默认使用 3D 视角。在 Editor Settings Inspector 中，可以随时修改此设置（2.5 节将介绍如何使用此工具）。

单击 Create project 按钮时，Unity 将在硬盘上生成项目，并在编辑器中打开该项目（如图 2-3 所示）。

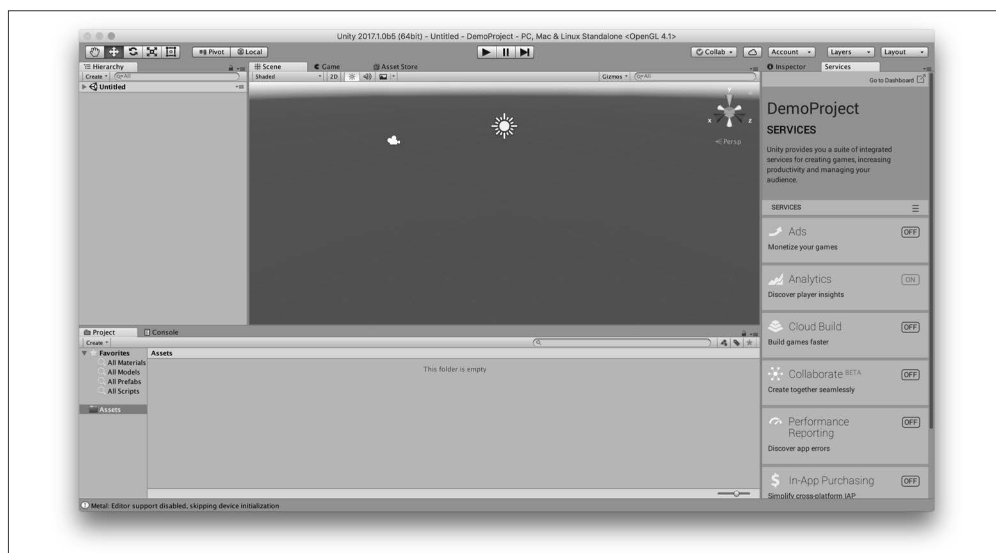
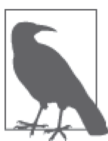


图 2-3：编辑器



项目结构

Unity 项目不是单独的文件，而是文件夹，其中包含 3 个重要的子文件夹：Assets、Library 和 ProjectSettings。Assets 文件夹包含游戏使用的所有文件：关卡、纹理、音效和脚本。Library 文件夹包含 Unity 内部使用的数据。ProjectSettings 文件夹中的文件包含项目设置。

一般不需要修改 Library 和 ProjectSettings 中的任何文件。

另外，如果使用了源代码控制系统，如 Git 或 Perforce，并不需要把 Library 文件夹签入存储库，但是需要签入 Assets 和 ProjectSettings 文件夹，以确保你的合作者使用的资源和设置与你的相同。

如果觉得这些内容有些陌生，完全可以置之不理，不过我们强烈建议你为代码使用合适的源代码控制标准，这会很有帮助。

Unity 采用了**窗格**设计。每个窗格的左上角有一个选项卡，可通过四处拖动这个选项卡来改变应用程序的布局。也可把某个选项卡拖出来，使其成为独立的窗口。默认情况下，并不是所有的窗格都是可见的，在构建游戏的过程中，将通过 Window 菜单打开更多的窗格。



如果你在开发过程中变得不知所措，那么可以打开 Window 菜单，然后选择 Layouts → Default 来重置布局。

Edit Mode和Play Mode

Unity 编辑器有两种模式：Edit Mode 和 Play Mode。在默认的 Edit Mode 中，可以创建场景、配置游戏对象以及构建游戏。在 Play Mode 中，可以玩游戏并与场景交互。

单击编辑器窗口上部的 Play 按钮可进入 Play Mode（如图 2-4 所示）。Unity 随即将启动游戏。要退出 Play Mode，只需再次单击 Play 按钮。



也可以按下 Command-P（PC 上为 Ctrl-P）组合键来进入和退出 Play Mode。



图 2-4：Play Mode 的控件

进入 Play Mode 后，可以按 Play Mode 控件中间的 Pause 按钮来暂停游戏。再次按下这个按钮将恢复游戏。按下最右侧的 Step 按钮，可以要求 Unity 前进一帧，然后再次暂停游戏。



退出 Play Mode 时，对场景做出的修改将被撤消。这包括玩游戏导致的修改，以及由于没有意识到自己在 Play Mode 中而对游戏对象做出的修改。进行修改前，一定要仔细确认自己在什么模式中。

接下来详细介绍默认显示的选项卡。本章按照窗格在默认布局中的位置进行介绍。（如果看不到某个窗格，请确认自己使用的是默认布局。）

2.2　场景视图

场景视图是窗口中部的窗格。你将在场景视图中投入大部分时间，因为在这里能够查看游戏**场景**的内容。

Unity 项目划分成多个场景。每个场景包含一组游戏对象，通过创建和修改游戏对象，就创建了游戏的世界。



可以把场景想象成关卡，但是场景还用于把游戏分解成可管理的块。例如，游戏主菜单，以及游戏的每个关卡，通常都有自己的场景。

2.2.1　模式选择器

场景视图有 5 种模式。窗口左上角的模式选择器（如图 2-5 所示）控制着与场景视图交互的方式。



图 2-5：场景视图的模式选择器，这里选择了 Translation 模式

从左到右，5 种模式介绍如下。

Grab 模式
: 选择此模式时，单击并拖动鼠标将移动视图。

Translation 模式
: 选择此模式时，可以移动当前选中的对象。

Rotation 模式
: 选择此模式时，可以旋转当前选中的对象。

Scale 模式
: 选择此模式时，可以调整当前选中对象的大小。

Rectangle 模式
: 选择此模式时，可以使用 2D 手柄移动当前选中的对象，还可以调整其大小。为 2D 场景进行布局，或者操作 GUI 时，这种模式特别有用。



在 Grab 模式中不能选择任何对象，但是在其他模式中可以。

使用模式选择器可改变场景视图的模式，也可以按下 Q、W、E、R 或 T 键在这些模式之间快速切换。

2.2.2　场景视图内的移动

有几种方法可在场景视图中四处移动。

- 单击窗口左上角的手形图标，进入 Grab 模式，然后通过单击和拖动来平移视图。
- 按下 Option 键（PC 上为 Alt 键），然后通过单击和拖动来旋转视图。
- 在场景中单击选中某对象，也可在 Hierarchy 中单击该对象的条目（2.3 节将进行讨论），然后将鼠标移动到场景视图上，按下 F 键可使视图集中到选中的对象上。
- 按下鼠标右键，然后移动鼠标，可四处查看场景。按住鼠标右键时，可以使用 W、A、S 和 D 键分别向前、左、后、右移动。也可以使用 Q 键和 E 键上下移动。按 Shift 键可加快移动速度。



也可以按 Q 键切换到 Grab 模式，而不必单击手形图标。

2.2.3 手柄控件

在模式选择器的右侧可找到手柄控件（如图 2-6 所示）。手柄控件决定了手柄——选择对象后显示的移动控件、旋转控件和缩放控件——的位置和方向。



图 2-6：手柄控件；图中，手柄的位置设为 Pivot，方向设为 Local

可配置的控件有两个：手柄的位置和方向。

手柄的位置可设置为 Pivot 或 Center。

- 设为 Pivot 时，手柄显示在对象的轴点。例如，人体 3D 模型的轴点通常位于两脚之间。
- 设为 Center 时，手柄显示在对象的中心，不考虑对象的轴点。

手柄的方向可设为 Local 或 Global。

- 设为 Local 时，手柄的方向相对于选中的对象。也就是说，如果旋转对象，使其“上”方向现在面向一侧，那么“上”箭头也将面向该侧。这使你能够沿着该对象的“本地”方向向上移动。
- 设为 Global 时，手柄的方向相对于世界。也就是说，“上”方向始终直指向上，不考虑对象的实际方向。当需要移动一个旋转的对象时，这个设置很有用。

2.3 Hierarchy窗格

在场景视图左侧可看到 Hierarchy（层次结构）窗格（如图 2-7 所示），其中显示了当前场景中的所有对象。如果场景很复杂，可利用这种层次结构，通过名称快速找到对象。

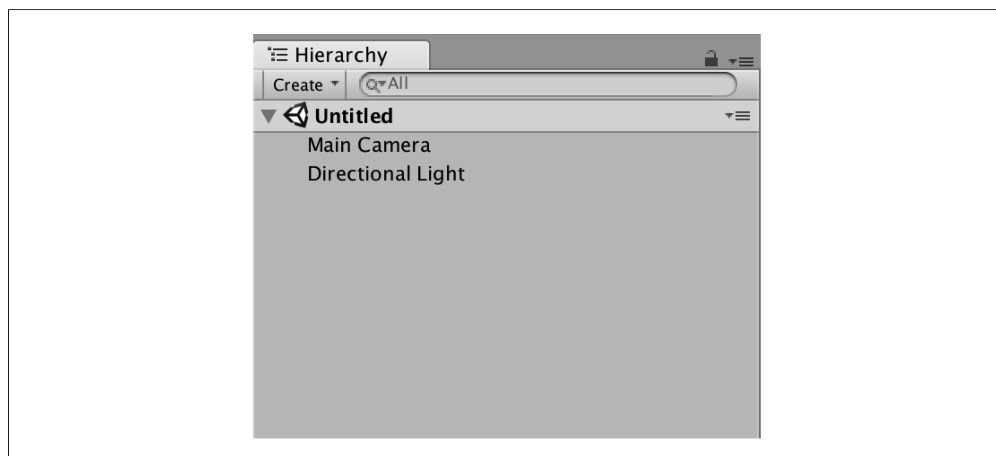


图 2-7：Hierarchy 窗格

顾名思义，在层次结构中也能看到对象的父子关系。在 Unity 中，对象可包含其他对象；在层次结构中，可浏览这种对象树。还可以通过拖放对象的方式在列表中重新排列它们。

层次结构的顶部有一个搜索框，在其中可输入想要寻找的对象的名称。在复杂的场景中，这种方法特别有用。

2.4 项目视图

项目视图（如图 2-8 所示）位于编辑器窗口的底部，显示了项目的 Assets 文件夹的内容。在项目视图中可使用游戏中的资源，以及管理文件夹布局。



移动、重命名和删除资源的操作应该只在项目视图中执行。这样，Unity 能够跟踪哪些文件发生变化。如果在项目视图之外执行这些操作（例如在 macOS 的 Finder 中，或者 PC 的 Windows 资源管理器中），Unity 将无法跟踪。这可能导致 Unity 无法正常处理资源，游戏也就无法正常工作。

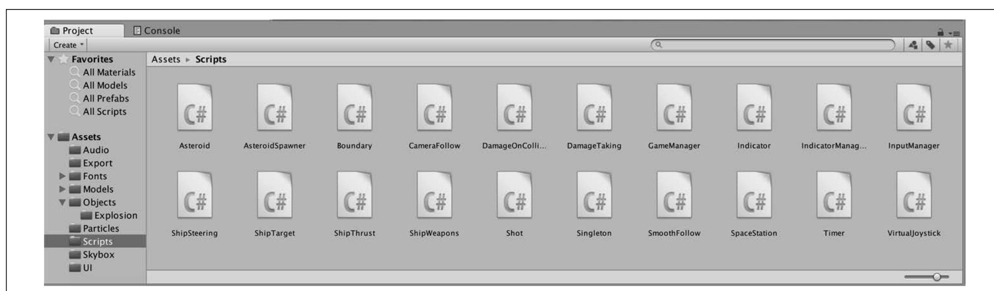


图 2-8：项目视图（这里显示了另外一个项目的资源；新创建的项目是空的）

项目视图可显示为单栏布局或双栏布局。图 2-8 显示了双栏布局；左栏显示了文件夹列表，右栏显示了当前选中的文件夹的内容。双栏视图最适合宽布局。与之不同，单栏视图（如图 2-9 所示）在一个列表中列出了所有的文件夹及其内容，非常适合较窄的布局。

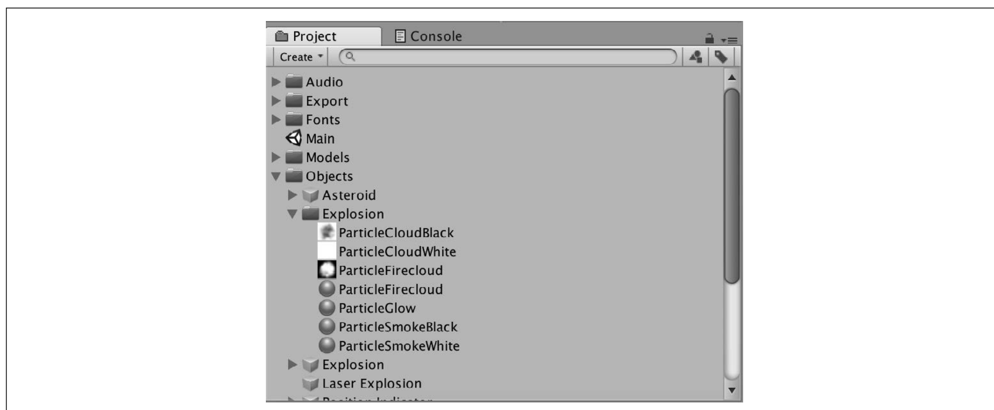


图 2-9：单栏模式的项目视图

2.5　Inspector

Inspector（如图 2-10 所示）是整个编辑器中最重要的视图之一，其重要程度仅次于场景视图。Inspector 显示了当前选中的对象的信息，也是配置游戏对象的地方。Inspector 显示在窗口的右侧；默认情况下，它与 Services 选项卡属于同一个选项卡组。

图 2-10：Inspector 显示了一个包含 Light 组件的对象的信息

Inspector 显示了与选中的对象或资源关联在一起的所有组件。每个组件显示不同的信息；第二部分和第三部分讲到构建项目时，我们将介绍各种各样的组件。随着内容的深入，我们将越来越熟悉 Inspector 及其内容。



除了显示当前选中对象的信息，Inspector 还显示了项目的设置。通过 Edit → Project Settings 菜单可访问项目设置。

2.6 游戏视图

游戏视图与场景视图在同一个选项卡组中，显示了游戏中当前活动的摄像机下看到的视图。进入 Play Mode（见 2.1 节）将自动激活游戏视图，允许你玩游戏。



游戏视图本身不具有交互性，它只是显示摄像机渲染的内容。这意味着当编辑器进入 Edit Mode 时，尝试与游戏视图交互不会有任何结果。

2.7 小结

了解了 Unity 的基本组成后，就可以使用它来完成自己想要的操作。对于这样复杂的软件，总会有许多要探索的地方；你应该花些时间四处看看。

第 3 章将介绍游戏对象和脚本。学完第 3 章后，我们就准备开发游戏了。

第3章

游戏中的脚本

要使游戏可玩，需要定义游戏中实际**发生**什么。Unity 提供了必要的基础功能，例如渲染图形、从玩家那里获取输入，以及播放音频。你需要做的就是添加自己的游戏所需要的功能。

怎么做呢？编写**脚本**，并将其添加到游戏的对象中。本章将介绍 Unity 的脚本系统，使用的编程语言是 C#。



Unity 中的语言

在 Unity 中编程时，有不同的语言可供选择。Unity 官方支持两种语言：C# 和“JavaScript”。

之所以把 JavaScript 放到引号中，是因为这里用到的实际上并不是你可能熟悉的那种 JavaScript 语言。相反，这是一种看起来像是 JavaScript 的语言，但是与 JavaScript 有许多区别。事实上，二者如此不同，以至于 Unity 的用户常常把这种语言叫作 UnityScript，甚至 Unity 团队有时候都采用这种叫法。

本书不使用 Unity 版本的 JavaScript，原因有两个。首先，Unity 的参考资料更多地展示了 C# 示例而不是 JavaScript 示例，这让我们感觉 Unity 的开发人员更倾向于使用 C#。

其次，在 Unity 中使用的 C# 与在其他地方使用的 C# 是一样的，但是 Unity 版本的 JavaScript 是专门用于 Unity 的。这意味着找到有关 C# 语言的帮助更加容易。

3.1 C#快速入门

我们使用 C# 为 Unity 游戏编写脚本。本书不会解释编程的基础知识（由于篇幅有限），不过会强调一些需要记住的要点。



Joseph Albahari 和 Ben Albahari 撰写的《果壳中的 C#——C# 5.0 权威指南》是关于 C# 语言的一本很好的参考书。

为了使你能够快速了解 C#，下面给出了一段 C# 代码，并标出了一些重要的元素：

```
using UnityEngine;  ❶

namespace MyGame{  ❷

    [RequireComponent(typeof(SpriteRenderer))]  ❸

    class Alien : MonoBehaviour{  ❹

        public bool appearsPeaceful;  ❺

        private int cowsAbducted;

        public void GreetHumans() {
            Debug.Log("Hello, humans!");

            if (appearsPeaceful== false) {
                cowsAbducted+= 1;
            }
        }
    }
}
```

❶ `Using` 关键字指定了想要使用的程序包。`UnitcyEngine` 程序包包含了 Unity 的核心类型。

❷ C# 允许把类型放到**名称空间**中，从而避免名称冲突。

❸ **特性**放在方括号内，允许添加关于类型或方法的额外信息。

❹ 使用 `class` 关键字定义**类**，在冒号后可指定超类。让一个类成为 `MonoBehaviour` 的子类后，就可以把该类作为一个脚本组件。

❺ 与类关联在一起的变量叫作**字段**。

3.2 Mono和Unity

Unity 的脚本系统由 Mono 框架提供支持。Mono 是微软的 .NET Framework 的开源实现，这意味着除了 Unity 自带的库，还可以使用 .NET 自带的完整的库集合。

认为 Unity 构建在 Mono 之上，是常见的误解。Unity 并不是构建于 Mono 之上，只是使用了 Mono 作为脚本引擎而已。借助 Mono，Unity 支持使用 C# 语言和 UnityScript 语言（Unity 版本的 JavaScript，参见本章开头“Unity 中的语言”）编写脚本。

Unity 中可用的 C# 和 .NET Framework 版本要比当前可用版本更早一些。作者于 2017 年初撰写本书时，可用的 C# 版本为 4，可用的 .NET Framework 版本为 3.5。其原因在于，Unity 使用自己的 Mono 项目分支，而这个分支在几年前就偏离了主流分支。这意味着 Unity 能够添加专供自己使用的功能，主要是一些针对移动方面的编译器功能。

目前，Unity 正在更新自己的编译器工具，让用户能够使用最新版本的 C# 和 .NET Framework。在更新完成之前，你的代码使用的版本会比最新版本落后几个版本。

因此，在网上查找 C# 代码或有关建议时，大部分时候应该搜索用于 Unity 的代码。与此类似，为 Unity 编写 C# 代码时，将混合使用 Mono 的 API（用于大部分平台均提供的通用内容）和 Unity 的 API（用于游戏引擎专用的内容）。

MonoDevelop

MonoDevelop 是 Unity 中包含的开发环境，主要用于编写脚本的文本编辑器，但是也包含一些有用的功能，可大大方便编程工作。

双击项目中的任意脚本文件时，Unity 将打开当前配置的编辑器，默认情况下即为 MonoDevelop，不过也可以配置为自己喜欢的其他任意文本编辑器。

Unity 会自动用项目中的脚本来更新 MonoDevelop 中的项目，并且当你返回 Unity 的时候会编译代码。也就是说，要编辑脚本，只需要保存修改并返回编辑器。

MonoDevelop 提供了一些能够大大节省时间的功能。

1. 代码完成

在 MonoDevelop 中，按 Ctrl- 空格键（PC 和 Mac 上均为此按键组合），MonoDevelop 将弹出一个窗口，显示了可能输入的内容的一个建议列表；例如，输入一个类名到一半时，MonoDevelop 会给出建议列表，按上下方向键可在列表中选择不同的建议项，按 Enter 键可接受建议。

2. 重构

按 Alt-Enter 组合键（Mac 上为 Option-Enter），MonoDevelop 将替你执行某些编辑源代码的任务。这些任务包括在 `if` 语句周围添加和删除括号，自动填充 `switch` 语句的 `case` 标签，或者将变量声明及变量赋值拆分到两行中。

3. 构建

回到编辑器时，Unity 会自动重新构建代码。不过，如果按 Command-B 组合键（PC 上的 F7 键），MonoDevelop 将构建所有代码。此操作产生的文件不会用在游戏中，但是执行此操作，意味着能够在返回 Unity 之前，确认代码中不存在编译错误。

3.3 游戏对象、组件和脚本

Unity 场景由**游戏对象**组成。游戏对象本身是不可见的对象，只不过有名称而已。游戏对象的行为由其**组件**定义。

组件是游戏的构建模块，你在 Inspector 中看到的所有东西都是组件。每个组件具有不同的职责；例如，Mesh Renderers 显示 3D 网格，而 Audio Sources 则向用户播放声音。你编写的脚本也是组件。

创建脚本的步骤如下。

(1) **创建脚本资源**。打开 Assets 菜单，选择 Create → Script → C# Script。
(2) **为脚本资源命名**。Project 面板的选定文件夹中将出现一个新的脚本文件，等待命名。
(3) **双击脚本资源**。该脚本将在脚本编辑器中打开，默认的脚本编辑器是 MonoDevelop。一开始，大部分脚本会类似于下面这样：

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class AlienSpaceship : MonoBehaviour {  ❶

    //进行初始化
    void Start () {  ❷

    }

    //每一帧调用一次Update
    void Update () {  ❸

    }
}
```

❶ 类的名称（在这里是 `AlienSpaceship`）必须与资源文件名相同。

❷ 第一次调用 `Update` 函数之前会调用 `Start` 函数。在 `Start` 函数中可添加代码来初始化变量、加载存储的首选项，或设置其他脚本和 GameObject。

❸ `Update` 函数在每一帧中都会被调用，可在其中添加代码来响应输入、触发另外一个脚本或者移动对象——也就是说，需要发生的任何事情都可以放在这里。



你可能熟知其他编程环境中的**构造函数**。在 Unity 中，不需要自己构建 `MonoBehaviour` 的子类，因为 Unity 自己会执行对象的构造，而且在你认为 Unity 可能会构造对象的时候，Unity 不一定会构造。

在附加到 GameObject 之前，Unity 中的脚本资源什么都不做，即它们的代码不会执行（如图 3-1 所示）。将脚本附加到 GameObject 的方法主要有两种。

(1) **将脚本资源拖放到 GameObject 上**。此操作可通过 Inspector 或 Hierarchy 窗格完成。
(2) **使用 Component 菜单**。在 Component → Scripts 下可找到项目中的所有脚本。

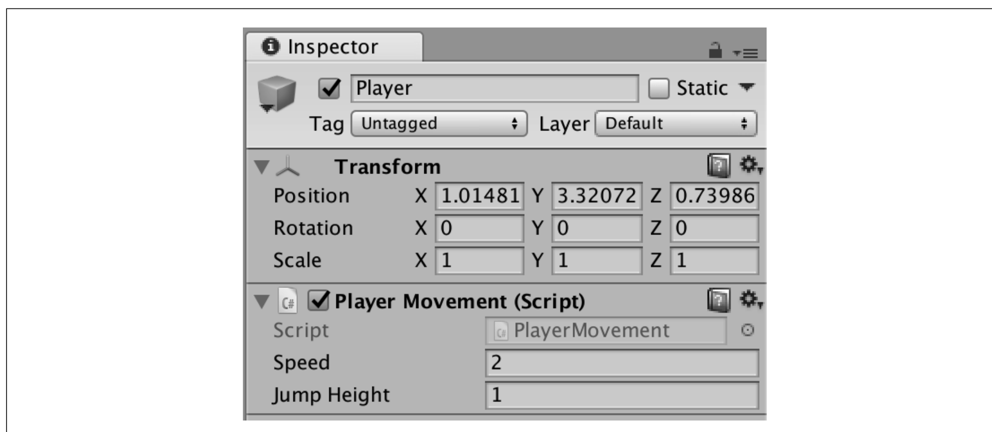


图 3-1：GameObject 的 Inspector，显示了作为组件添加的脚本“PlayerMovement”

由于在 Unity 的编辑器中，脚本主要通过作为组件附加到 GameObject 的方式公开，Unity 允许把脚本中的特性作为可编辑的值显示在 Inspector 中。为利用这种功能，需要在脚本中创建一个公共变量。只要设为“公共”，在编辑器中就是可见的。不过，也可以把变量设置为私有变量。

```
public class AlienSpaceship : MonoBehaviour {
    public string shipName;

    //Inspector中将显示一个可编辑的Ship Name字段
}
```

3.3.1　Inspector

当把脚本作为组件添加到游戏对象上以后，选中该游戏对象时，脚本就会出现在 Inspector 中。Unity 将按照在代码中出现的顺序，自动显示所有公共变量。

带有 [SerializeField] 特性的私有变量也会显示在 Inspector 中。当你想让某个字段显示在 Inspector 中，但是不想其他脚本能够访问该字段时，这种功能很有用。



在显示变量名时，Unity 编辑器将大写每个单词的首字母，然后在原有大写字母的前面加上一个空格。例如，变量 shipName 在编辑器中显示为“Ship Name”。

3.3.2　组件

通过使用 GetComponent 方法，脚本可访问 GameObject 中的不同组件：

```
//如果此对象有Animator组件，则获取该Animator组件
var animator = GetComponent<Animator>();
```

也可以调用其他对象的 GetComponent 方法来获取这些对象附加的组件。

通过使用 GetComponentInChildren 和 GetComponentInParent 方法，还可以获取附加到子对象和父对象的组件。

3.4 重要的方法

MonoBehaviours 类中包含几个对于 Unity 特别重要的方法。这些方法在组件生命周期的不同时刻调用，可用于在适当的时刻运行适当的行为。本节按照这些方法的运行顺序来介绍它们。

3.4.1 Awake和OnEnable

在场景中实例化某个对象以后，会立即运行 Awake，这是运行脚本代码的第一个机会。在对象的生命周期中，只会调用 Awake 方法一次。

与之相对的是，每当激活一个对象的时候，就会调用 OnEnable 方法。

3.4.2 Start

在第一次调用对象的 Update 方法之前，将调用 Start 方法。

对比Start方法与Awake方法

你可能在想，为什么 Unity 提供了 Awake 方法和 Start 方法来设置对象，是不是说可以随意选择一个方法来进行设置呢？

事实上，之所以提供两个方法，是有很好的理由的。启动一个场景时，场景中的所有对象都会运行 Awake 和 Start 方法。但是，重要的地方在于，Unity 确保在运行任何 Start 方法**之前**，所有对象的 Awake 方法已经完成运行。

这意味着，当一个对象运行其 Start 方法时，其他对象的 Awake 方法一定已经完成了工作。这一点可能很有用，例如在下例中，ObjectA 使用了 ObjectB 建立的一个字段：

```
//保存在文件ObjectA.cs中
class ObjectA : MonoBehaviour {

    //供其他脚本访问的变量
    public Animator animator;

    void Awake() {
        animator = GetComponent<Animator>();
    }
}

//保存在ObjectB.cs文件中
class ObjectB : MonoBehaviour {

    //连接到ObjectA脚本
    public ObjectA someObject;

    void Awake() {
```

```
        //检查someObject是否设置了animator变量
        bool hasAnimator = someObject.animator == null;

        //取决于哪一个方法先调用，可能输出true或false
        Debug.Log("Awake: " + hasAnimator.ToString());
    }

    void Start() {
        //检查someObject是否设置了animator变量
        bool hasAnimator = someObject.animator == null;

        //始终输出true
        Debug.Log("Start: " + hasAnimator.ToString());
    }
}
```

在本例中，包含 `ObjectA` 脚本的对象还附加了一个 Animator 组件（本例中的 Animator 什么都不做，所以也可以是任何其他类型的组件）。`ObjectB` 脚本的 `someObject` 变量连接到包含 `ObjectA` 脚本的对象上。

当场景开始时，`ObjectB` 脚本将记录两次：一次在 `Awake` 方法中，另一次在 `Start` 方法中。在这两种情况中，它将试图确定 `someObject` 变量的 `animator` 字段是不是为空，然后输出 `true` 或 `false`。

如果运行本示例，那么第一条消息（在 `ObjectB` 的 `Awake` 方法中运行）可能是 `true`，也可能是 `false`，取决于哪个脚本的 `Awake` 方法先运行。这是因为在 Unity 中，如果没有手动设置执行顺序，那么根本无法知道哪个 `Awake` 方法先运行。

但是，在 `ObjectB` 的 `Start` 方法中运行的第二条消息**一定**会返回 `true`。这是因为当场景启动时，所有已存在对象的 `Awake` 方法都会运行完成，之后才会运行 `Start` 方法。

3.4.3 Update和LateUpdate

只要启用了组件，并且附加脚本的对象是激活的，那么 `Update` 方法在每一帧中就都会被调用。



因为每一帧都会运行 `Update` 方法，所以应该让 `Update` 方法做尽可能少的工作。如果在 `Update` 方法中执行耗时的工作，会拖慢游戏的其余部分。如果需要执行一些耗时的工作，应该使用协程（下一节将进行讨论）。

Unity 将调用所有脚本的 `Update` 方法（前提是脚本有 `Update` 方法）。然后，将调用所有脚本的 `LateUpdate` 方法（前提是脚本有 `LateUpdate` 方法）。`Update` 和 `LateUpdate` 之间的关系类似于 `Awake` 和 `Start` 的关系：直到运行完所有 `Update` 方法后，才会调用 `LateUpdate` 方法。

当想要执行的工作依赖于在 `Update` 方法中完成工作的其他对象的时候，这一点很有用。你无法控制哪个对象首先运行 `Update` 方法，但是在 `LateUpdate` 方法中写代码的时候，可以确信全部对象的 `Update` 方法都已完成运行。



除了 Update 方法，还可以使用 FixedUpdate 方法。Update 方法在每一帧中调用一次，而 FixedUpdate 方法则在每一秒中调用固定次数。当用到物理时，需要每隔一定的时间段施加外力，这时候 FixedUpdate 方法很有用。

3.5 协程

大部分函数在完成工作后就立即返回。然而，有些时候需要让一些工作用些时间逐渐完成。例如，如果想让一个对象从一个位置滑动到另一个位置，就需要让这种移动发生在多个帧中。

在多个帧中运行的函数称为**协程**。要创建协程，首先需要创建一个返回类型为 IEnumerator 的方法：

```
IEnumerator MoveObject() {

}
```

接下来，使用 yield return 语句让协程临时停止运行，使游戏的其余部分能够继续执行。例如，要使一个对象在每一帧中向前移动一定距离[1]，可以使用下面的代码：

```
IEnumerator MoveObject() {
    //无限循环下去
    while (true) {

        transform.Translate(0,1,0); //每一帧在y轴上移动一个单位

        yield return null;//等待进入下一帧
    }
}
```



如果包含一个无限循环（例如上例中的 while(true)），那么在循环期间**必须**使用 yield 让出执行权。否则，循环将一直执行，其他代码将没有机会执行其他工作。由于游戏的代码运行在 Unity 内，如果进入无限循环，可能导致 Unity 不能响应。如果发生这种情况，需要强行关闭 Unity，而这可能导致丢失未保存的工作。

从一个协程 yield return 时，将暂时停止执行该函数。Unity 将在以后恢复执行该函数；具体**何时**恢复执行取决于使用什么值 yield return。

例如：

yield return null
等到下一帧恢复执行

yield return new WaitForSeconds(3)
等待 3 秒后恢复执行

注 1：这么做实际上并不是一个好主意，3.8 节将解释原因。举这个例子是因为它是一个最简单的例子。

yield return new WaitUntil(() => **this**.someVariable == **true**)

等待 someVariable 等于 true 时执行；在这里可以使用任何计算结果为 true 或 false 变量的表达式



要停止执行协程，需要使用 yield break 语句：

```
//立即停止此协程
yield break;
```

当执行到方法末尾时，协程也会自动停止执行。

有了协程函数后，就可以启动该函数。要启动一个协程，不能单独进行调用，而要使用 StartCoroutine 函数进行调用：

```
StartCoroutine(MoveObject());
```

启动协程后，它将开始执行，直到遇到 yield break 语句，或者到达函数末尾。



除了刚刚提到的 yield return 示例，还可以 yield return 另一个协程。这意味着该协程将等待另一个协程结束。

在协程外部也可以停止一个协程。要使用这种方法，需要保留对 StartCoroutine 方法的返回值的一个引用，并将其传递给 StopCoroutine 方法：

```
Coroutine myCoroutine = StartCoroutine(MyCoroutine());

//……后面……

StopCoroutine(myCoroutine);
```

3.6 创建和销毁对象

在游戏运行期间，有两种方法可创建对象。第一种方法是创建一个空的 GameObject，然后使用代码给该 GameObject 附加组件；第二种方法是复制另一个对象（称为**实例化**）。第二种方法可在一行代码中完成所有设置，所以更受欢迎。我们首先讨论这种方法。



在 Play Mode 中创建新对象时，这些对象将在停止游戏后消失。如果想要这些对象保留下来，需要执行下列步骤。

(1) 选择想要保存的对象。

(2) 按 Command-C（在 PC 上按 Ctrl-C）键，或者打开 Edit 菜单并选择 Copy 命令，复制这些对象。

(3) 退出 Play Mode。对象将从场景中消失。

(4) 按 Command-V（在 PC 上按 Ctrl-V）键，或者打开 Edit 菜单并选择 Paste，粘贴前面复制的对象。对象将重新出现，现在就可以在 Edit Mode 中操作这些对象了。

3.6.1　实例化

在 Unity 中，实例化一个对象意味着复制该对象，以及该对象的所有组件、子对象以及**子对象**的组件。当实例化的对象是一个预设（prefab）时，这一点尤为强大。预设是作为资源保存的预构建对象。这意味着你可以创建对象的一个模板，然后在许多不同的场景中实例化该模板的多个副本。

要实例化一个对象，需要使用 `Instantiate` 方法：

```
public GameObject myPrefab;

void Start() {
    //创建myPrefab的一个新副本，将其置于与此对象相同的位置
    var newObject = (GameObject)Instantiate(myPrefab);

    newObject.transform.position = this.transform.position;
}
```



`Instantiate` 方法的返回类型是 `Object`，而不是 `GameObject`。需要执行转换来将其作为 `GameObject` 处理。

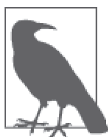
3.6.2　从头创建对象

创建对象的另外一种方法是通过代码自己创建。为此，需要使用 `new` 关键字构造一个新的 `GameObject`，然后调用该 `GameObject` 的 `AddComponent` 方法来添加新组件：

```
//创建一个新的游戏对象
//在Hierarchy中，新游戏对象将显示为My New GameObject
var newObject = new GameObject("My New GameObject");

//向游戏对象添加一个新的SpriteRenderer
var renderer = newObject.AddComponent<SpriteRenderer>();

//告诉新的SpriteRenderer显示一个精灵
renderer.sprite = myAwesomeSprite;
```



`AddComponent` 方法接受一个泛型参数，即想要添加的组件类型。在这里可以指定 `Component` 类的任意子类，该组件将被添加到 `GameObject` 中。

3.6.3　销毁对象

`Destroy` 方法从场景中移除对象。注意，这里的用词是“对象”（object），而不是“游戏对象”（game object）。`Destroy` 方法既可以移除游戏对象，又可以移除组件。

要从场景中移除游戏对象，需要对该对象调用 `Destroy` 方法：

```
//销毁此脚本关联到的游戏对象
Destroy(this.gameObject);
```



Destroy 方法可操作组件和游戏对象。

如果在调用 Destroy 方法时传入 this（代表当前的脚本组件），并不会移除游戏对象；相反，脚本将从所附的游戏对象上移除。游戏对象仍将存在，但是不再附有你的脚本。

3.7 特性

特性（attribute）是可以附加到类、变量或方法上的一条信息。Unity 定义了几种有用的特性，可改变类的行为或者类在编辑器中呈现的方式。

1. RequireComponent

当附加到类时，RequireComponent 特性允许告知 Unity，此脚本要求另外一种类型的组件必须存在。当脚本只有附加了该类型的组件才有意义时，这个特性会很有用。例如，如果脚本只做一件事，比如修改 Animator 的设置，那么类要求必须存在一个 Animator 是很合理的。

为了指定某个组件要求必须存在的组件类型，需要提供该组件类型作为参数，例如：

```
[RequireComponent(typeof(Animator))]
class ClassThatRequiresAnAnimator : MonoBehaviour {
    //此类要求GameObject上关联一个Animator
}
```



如果添加的脚本要求 GameObject 有特定的组件，但是 GameObject 还没有该组件，那么 Unity 将自动把该组件添加到 GameObject 中。

2. Header和Space

把 Header 特性添加到一个字段时，Unity 会在 Inspector 中在该字段的上方绘制一个标签。Space 的工作方式与此类似，只不过是添加一个空行。二者对于组织 Inspector 的内容都很有用。

例如，图 3-2 显示了下列代码在 Inspector 中的显示效果：

```
public class Spaceship : MonoBehaviour {

    [Header("Spaceship Info")]

    public string name;

    public Color color;

    [Space]

    public int missileCount;
}
```

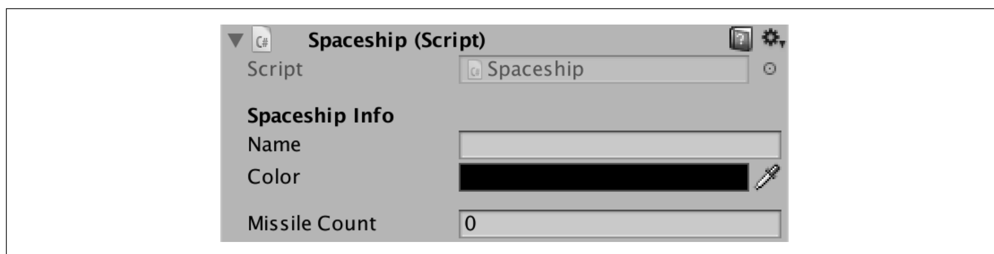


图 3-2：显示标签和空行的 Inspector

3. SerializeField和HideInInspector

正常情况下，只有公共字段才会显示在 Inspector 中。但是，将变量声明为 `public`，意味着其他对象能够直接访问它们；这样一来，对象就很难完全控制自己的数据。然而，如果将变量声明为 `private`，Unity 就不会在 Inspector 显示该变量。

为了处理这个问题，当想要在 Inspector 中显示一个私有变量的时候，可以向该变量添加 `SerializeField` 特性。

如果想要获得**相反**的行为（即，变量是 `public` 变量，但是不显示在 Inspector 中），那么可以使用 `HideInInspector` 特性：

```
class Monster : MonoBehaviour {

    //由于是公有的，所以会显示在Inspector中
    //可在其他脚本中访问
    public int hitPoints;

    //因为是私有的，所以不会显示在Inspector中
    //在其他脚本中无法访问
    private bool isAlive;

    //由于设置了SerializeField，所以会显示在Inspector中
    //在其他脚本中无法访问
    [SerializeField]
    private int magicPoints;

    //由于设置了HideInInspecctor，所以不会显示在Inspector中
    //可在其他脚本中访问
    [HideInInspector]
    public bool isHostileToPlayer;
}
```

4. ExecuteInEditMode

默认情况下，脚本只在 Play Mode 中运行自己的代码。也就是说，只有当游戏实际运行的时候，`Update` 方法才会运行。

但是，有时候让代码一直运行会很方便。这种情况下，可以向类添加 `ExecuteInEditMode` 特性。



组件的生命周期在 Edit Mode 下和在 Play Mode 下有所不同。在 Edit Mode 下，Unity 只在必要的时候重绘，通常是为了响应用户输入事件（如鼠标单击）。这意味着 `Update` 方法只是间或运行，而不是连续运行。另外，协程的行为会与你的预期不同。

另外，在 Edit Mode 下不能调用 `Destroy`，因为 Unity 的行为是推迟到下一帧才实际移除对象。在 Edit Mode 下，应该调用 `DestroyImmediate`，该方法将立即移除对象。

例如，下面的脚本使一个对象始终面向其目标，即使当前不在 Play Mode 下：

```
[ExecuteInEditMode]
class LookAtTarget : MonoBehaviour {

    public Transform target;

    void Update() {
        //如果没有目标，就不继续执行
        if (target != null) {
            return;
        }

        //转动以面对目标
        transform.LookAt(target);
    }

}
```

如果将这个脚本附加到一个对象，并将其 `target` 变量设为另外一个对象，那么不管是在 Play Mode 还是 Edit Mode 下，第一个对象将转动自身来面向其目标。

3.8 脚本中的时间

`Time` 类用来在游戏中获取关于当前时间的信息。`Time` 类中有几个变量（强烈建议你查阅相关文档，https://docs.unity3d.com/Manual/TimeFrameManagement.html），但是最重要、最常用的变量是 `deltaTime`。

`Time.deltaTime` 计算上一帧被渲染后到现在的时间。计算出的时间可能有很大的波动，认识到这一点很重要。使用此变量时，可以执行在每一帧更新、但是需要一定时间才能完成的动作。

3.5 节的示例在每一帧中将对象移动一个单位。这么做不是一个好主意，因为一秒中的帧数可能发生很大变化。例如，如果摄像机对准场景中的一个非常简单的部分，每秒的帧数可能非常高，而当摄像机对准更加复杂的场景时，帧率可能很低。

因为不能确定当前运行游戏时每秒的帧数，所以最好将 `Time.deltaTime` 作为考虑因素。用一个例子来解释这一点，会比较容易理解：

```
IEnumerator MoveSmoothly() {
    while (true) {

        //每秒移动一个单位
        var movement = 1.0f * Time.deltaTime;

        transform.Translate(0, movement, 0);

        yield return null;

    }
}
```

3.9 记录到控制台

在 3.4.1 节我们看到，有时候把一些信息记录到控制台比较方便，例如为了方便诊断问题，或者提醒自己存在某个问题。

`Debug.Log` 函数可实现此目的。日志记录有 3 个级别：信息、警告和错误。这 3 种类型没有功能上的区别，只不过警告和错误更加醒目。

除了 `Debug.Log`，还可以使用 `Debug.LogFormat`，在发送给控制台的字符串中嵌入值：

```
Debug.Log("This is an info message!");
Debug.LogWarning("This is a warning message!");
Debug.LogError("This is a warning message!");

Debug.LogFormat("This is an info message! 1 + 1 = {0}", 1+1);
```

3.10 小结

编写脚本是 Unity 中的一项重要技能，熟悉 C# 语言和编写 C# 脚本的工具后，构建游戏将变得更加容易和有趣。

第二部分

构建2D游戏：地精寻宝

大体了解了 Unity 之后，我们将实际运用学到的技能。在第二部分和第三部分，我们将从头构建完整的游戏。

在接下来的几章中，我们将构建一个卷轴动作游戏，命名为 *Gnome's Well That Ends Well*。此游戏在很大程度上依赖于 Unity 的 2D 图形和物理功能，另外还大量使用了 UI 系统。构建这个游戏会很有趣。

第 4 章

开始构建游戏

知道 Unity 界面各部分的功能是一回事，使用这个界面创建完整的游戏是另一回事。在第二部分，我们将运用第一部分学到的知识来创建一个 2D 游戏。到第二部分结束时，我们将创建完成一个卷轴动作游戏：*Gnome's Well That Ends Well*（图 4-1 是该游戏的一个截图）。

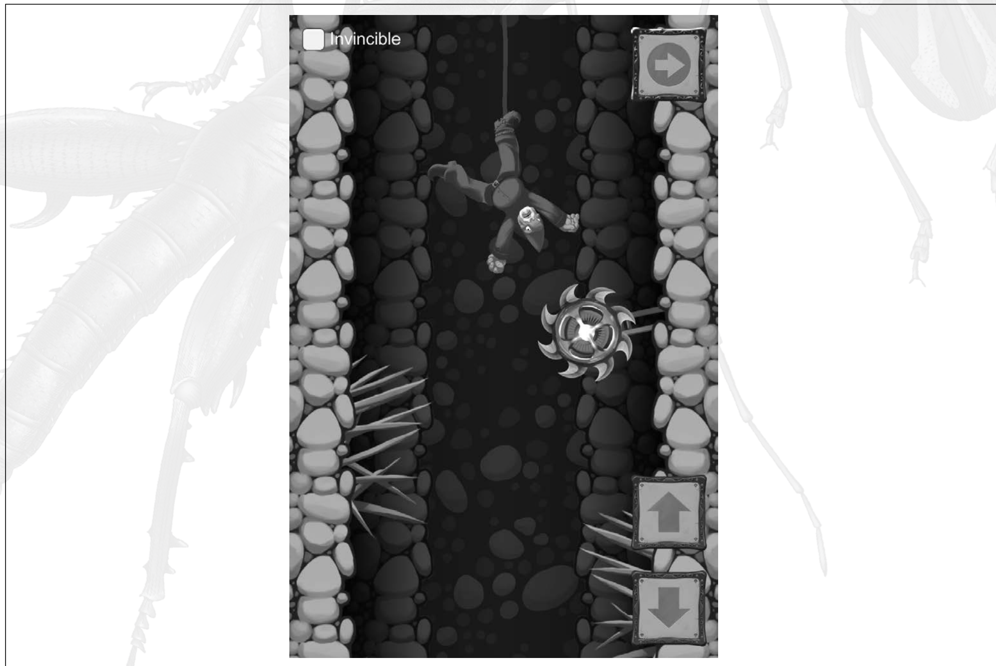


图 4-1：最终的游戏效果（另见彩插）

4.1 游戏设计

Gnome's Well 的玩法很简单。玩家控制一个花园地精，用一根绳索将其坠到一个井中，井底有宝藏。井壁上有一些陷阱，地精碰到陷阱后就会死亡。

首先创建一个非常粗糙的草图，展示游戏将会是什么样子。我们使用优秀的绘图程序 OmniGraffle，但是选择什么工具并不是特别重要，使用纸和笔一样很简单，而且很多时候效果更好。这里的目标是尽快形成一个大概的想法，知道如何把游戏的各个部分组织起来。图 4-2 显示了 *Gnome's Well* 的草图。

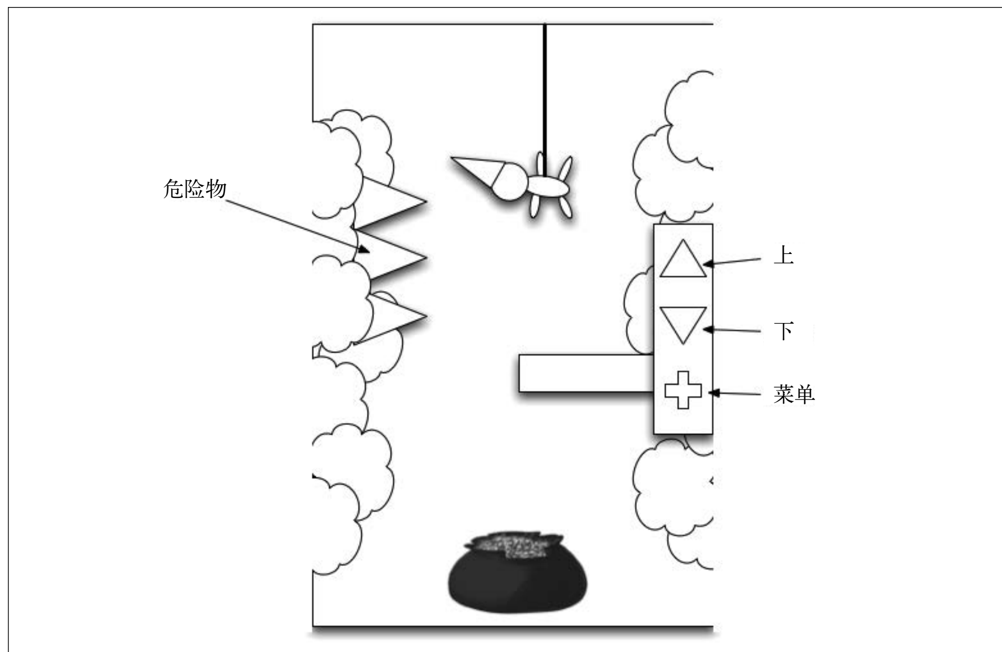


图 4-2：游戏的简略概念图

确定了游戏的最终效果后，还需要确定总体架构。首先确定哪些对象是可见的，以及它们之间的关系是什么。同时，我们思考"不可见的"组件如何工作：如何收集输入信息，游戏的内部管理器如何彼此通信。

最后，还要考虑游戏的视觉效果。我们找到了一个艺术家朋友，请他绘制一幅地精下到井中、遭遇陷阱威胁的图片。这让我们意识到主要角色可能是什么样子，并设定了游戏的总体基调：轻松的、卡通风格的、稍微有点暴力的游戏，主角是贪婪的地精。最终概念图如图 4-3 所示。



如果不认识艺术家，就自己绘图！也许你觉得自己画得不好，但是只要能表达游戏看起来是什么样子，**任何**想法都比没有想法要好。

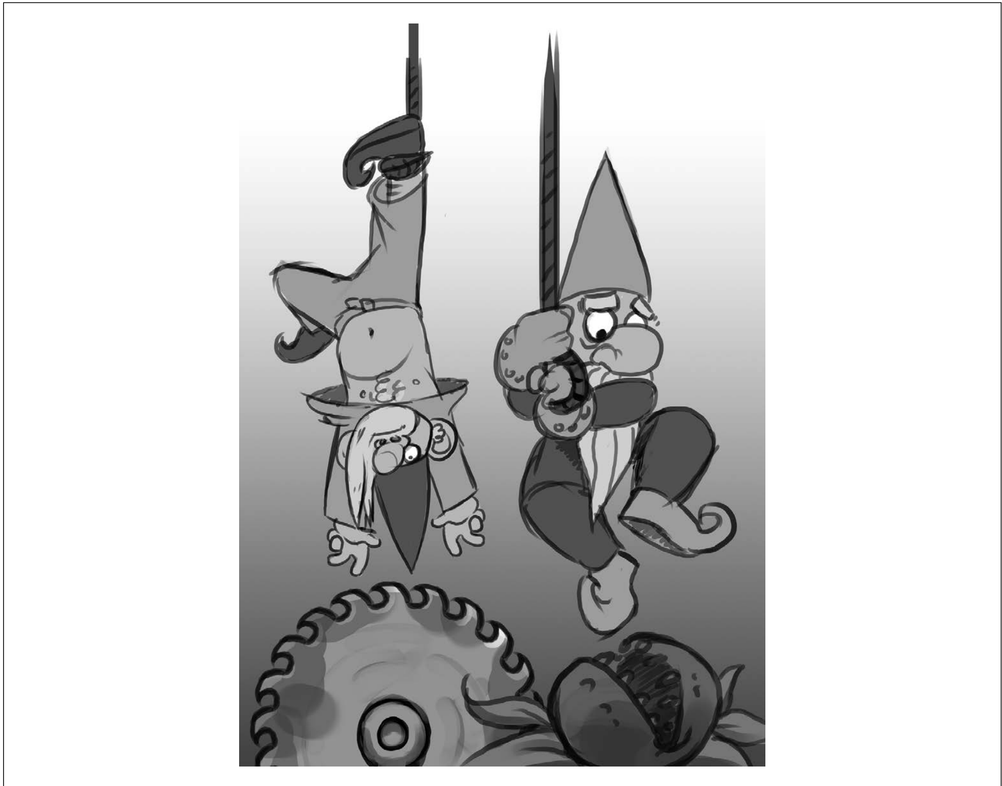


图 4-3：地精角色的概念图

完成了初步设计后，就能够开始确定一些需要实现的东西：地精在游戏中如何移动，如何设置界面，以及游戏对象如何联系起来。

为使地精能够下到井中，可给玩家提供 3 个按钮：一个增加绳索的长度，一个缩短绳索的长度，还有一个显示游戏的菜单。按下增加绳索长度的按钮，地精将在井中下降。为了在下降过程中避开陷阱，玩家需要左右倾斜设备，使地精左右移动。

游戏玩法主要是 2D 物理模拟的结果。地精是一个“布娃娃”（ragdoll），即通过关节连接的部位集合，每个部位是一个独立模拟的刚体。这意味着当通过 Rope（绳索）对象连接到井的顶部时，地精将正确摆动。

绳索是通过类似的方式创建的：绳索是刚体的集合，各个刚体通过关节彼此连接。刚体链中的第一段连接到井口，并通过一个旋转的关节连接到第二段。第二段连接到第三段，第三段连接到第四段，以此类推，直到最后一段连接到地精的脚踝。要延长绳索，就在绳索顶部添加更多绳段；要缩短绳索，就移除绳段。

游戏玩法的剩余部分通过非常直观的碰撞检测来处理。

- 如果地精的任何部位触碰到陷阱对象，地精将死亡，然后创建一个新的地精。另外，创建一个鬼魂精灵，沿着井向上移动。

- 如果触碰到宝藏，就更新地精的精灵，显示它抱着宝藏。
- 如果触碰到井口（不可见对象），并且地精抱着宝藏，那么玩家获胜。

除了地精、陷阱和宝藏，游戏摄像机的脚本也在运行，使摄像机的位置关联到地精的纵向位置，但也会阻止摄像机显示高于井口或者低于井底的任何东西。

我们构建这个游戏的方式如下（不必担心，我们将带着你逐步完成）。

(1) 首先，暂时使用简单的火柴人来创建地精。我们将设置布娃娃，并连接精灵。
(2) 接下来，设置绳索。这里涉及第一段大块代码，因为绳索是在运行时生成的，并且需要支持绳索的延长和缩短。
(3) 设置好绳索后，将创建输入系统。输入系统将接收关于设备如何倾斜的信息，并将此信息提供给游戏的其他部分（特别是地精）使用。同时，我们将构建游戏的用户界面，并创建延长和缩短绳索的按钮。
(4) 创建好绳索、地精和输入系统后，就可以开始实际创建游戏了。我们将实现陷阱和宝藏，并开始玩游戏。
(5) 剩下的就是锦上添花了：将地精的精灵替换为更加复杂的精灵，添加粒子效果，以及添加整体音效。

到本章结束时，游戏功能已经完整，但是有些艺术效果还没有完善。第 7 章将完成这部分内容。图 4-4 显示了本章结束时的游戏效果。



图 4-4：第一个粗糙版本完成后的游戏效果



在完成这个项目的过程中，我们将向游戏对象添加大量组件，并调整属性值。除了我们告诉你需要修改的组件之外，还有大量组件可以调整，因此你可以自由修改任何东西的设置来查看效果，否则就保留默认设置。

接下来就正式开始游戏的构建！

4.2　创建项目并导入资源

我们首先在 Unity 中创建项目，并完成一些设置工作。然后导入早期阶段需要的一些资源。随着过程的深入，我们将导入更多资源。

(1) **创建项目**。选择 File → New Project，创建一个新项目，命名为 GnomesWell。在 New Project 对话框中（如图 4-5 所示），注意选择 2D 而不是 3D，并确保没有导入任何资源包。现在我们只是想创建一个空项目。

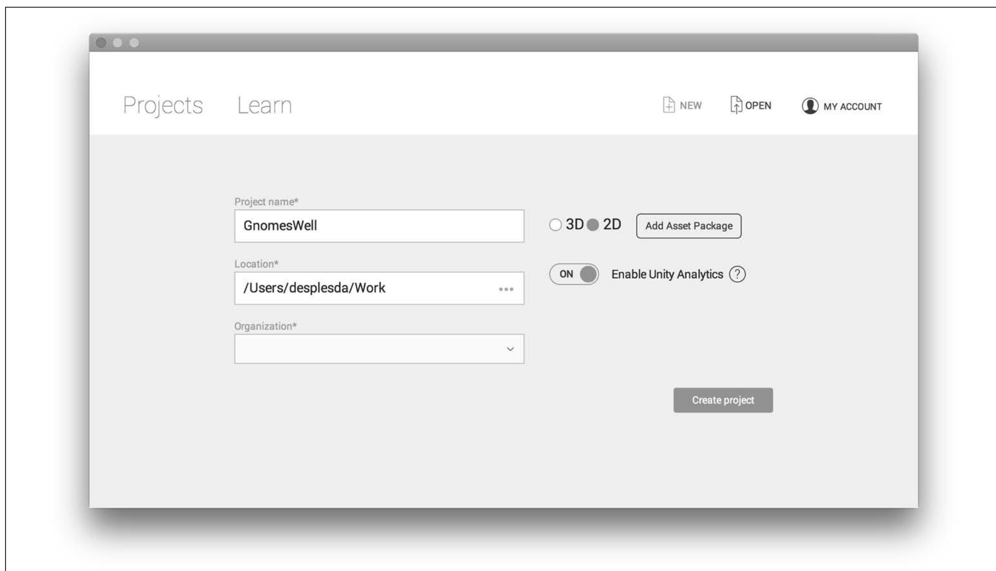


图 4-5：创建项目

(2) **下载资源**。我们已经打包了此项目需要的图片、声音和其他资源，可从 https://www.secretlab.com.au/books/unity 下载。[1] 下载完成后，解压到合适的文件夹。后面将把这些资源导入到项目中。

(3) **将场景保存为 Main.scene**。最好现在就保存场景，这样以后按 Command-S（PC 上为 Ctrl-S）键时将立即保存工作。第一次保存时，需要提供场景的名称和保存位置，这里把场景保存到 Assets 文件夹中。

注 1：也可从本书图灵社区页面（http://www.ituring.com.cn/book/2117）的“随书下载”处点击下载。
——编者注

(4) **创建项目中的文件夹**。为了让工作井井有条，为不同类别的资源创建不同的文件夹是一个好主意。Unity 支持把所有内容放在一个文件夹中，但是这么做会使得查找资源非常麻烦。在 Project 选项卡中右击 Assets 文件夹，选择 Create → Folder，创建下面的文件夹。

Scripts
　此文件夹将包含游戏中的 C# 代码。（默认情况下，Unity 会将新的代码文件放到根目录 Assets 中，你需要自己把它们移动到 Scripts 文件夹中。）

Sounds
　此文件夹将包含音乐和音效。

Sprites
　此文件夹将包含所有精灵图像。精灵图像有很多，所以要把它们放到子文件夹中。

Gnome
　此文件夹将包含地精角色需要的预设，以及其他相关的对象，例如绳索、粒子效果和鬼魂。

Level
　此文件夹将包含关卡自身的预设，包括背景、井壁、装饰用的对象和陷阱。

App Resources
　此文件夹将包含应用作为一个整体需要的资源：应用图标及其闪屏。

完成创建文件夹的工作后，Assets 文件夹应该如图 4-6 所示。

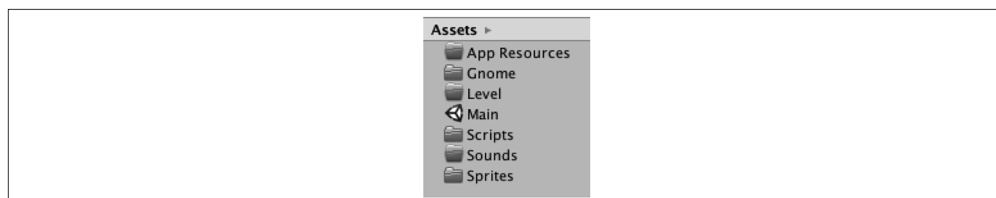


图 4-6：创建文件夹之后的 Assets 文件夹

(5) **导入原型地精资源**。原型地精是地精的粗糙版本。我们首先构建这个版本，后面将把它替换为更加美观的精灵。

在下载的资源中，找到 Prototype Gnome 文件夹，将其拖放到 Unity 的 Sprites 文件夹中，如图 4-7 所示。

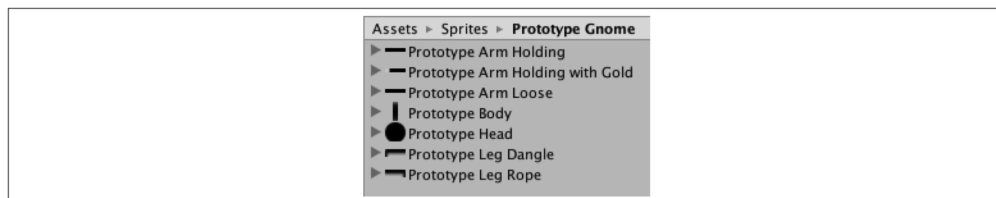


图 4-7：地精的原型精灵

现在就可以开始构造地精了。

4.3 创建地精

因为地精将由多个独立移动的对象组成，所以我们需要先创建一个对象，作为每个身体部位的容器。还需要给此对象添加 Player 标签，因为用来检测地精何时触碰到陷阱、宝藏或者关卡出口的碰撞检测系统需要知道该对象是否为特殊的 Player 对象。构建地精的步骤如下。

(1) **创建 Prototype Gnome 对象**。打开 GameObject 菜单并选择 Create Empty，创建一个新的空游戏对象。

将新对象命名为 Prototype Gnome，然后从 Inspector 顶部的 Tag 下拉列表中选择 Player，将该对象的标签设为 `Player`。



Prototype Gnome 对象的 Position 在 *x*、*y* 和 *z* 轴上均应该是 0，从 Inspector 顶部的 Transform 组件可看到这一点。如果不在位置 0，可以单击 Transform 组件右上角的齿轮菜单，选择 Reset Position 命令。

(2) **添加精灵**。找到前面添加的 Prototype Gnome 文件夹，将每个精灵拖放到场景中，但是不要拖放 Prototype Arm Holding with Gold，这个精灵以后才会用到。



你需要单独拖放每个精灵。如果选择所有精灵，试图把它们一次全部拖放到场景中，那么 Unity 会认为你在试图拖放一系列图像，从而创建一个动画。

完成拖放操作后，场景中将有 6 个新精灵：Prototype Arm Holding、Prototype Arm Loose、Prototype Body、Prototype Head、Prototype Leg Dangle 和 Prototype Leg Rope。

(3) **将精灵设置为 Prototype Gnome 对象的子对象**。在 Hierarchy 窗格中，选择刚才添加的全部精灵，把它们拖放到空的 Prototype Gnome 对象上。完成之后，Hierarchy 应该如图 4-8 所示。

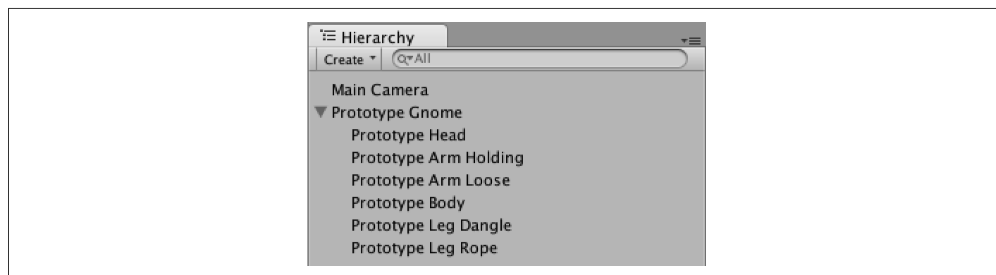


图 4-8：在 Hierarchy 中，地精精灵已被附加为 Prototype Gnome 对象的子对象

(4) **设置精灵的位置**。添加精灵后，需要正确设置它们的位置，把胳膊、腿和头连接到躯干上。在场景视图中，单击工具栏中的 Move 工具或者按下 T 键，选择 Move 工具。

使用 Move 工具重新排列精灵，使其如图 4-9 所示。

另外，让全部精灵使用 Player 标签，就像其父对象一样。最后，确保每个对象的 *z* 位置为 0。在每个对象的 Transform 组件（位于 Inspector 顶部）的 Position 字段中，可看到其 *z* 位置。

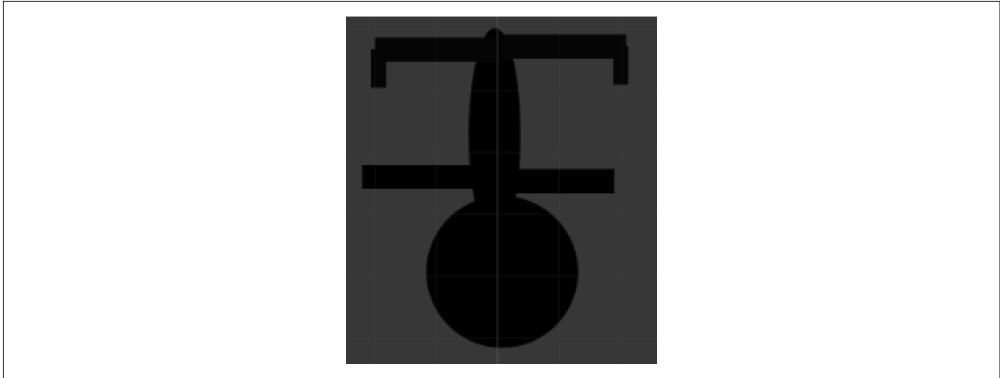


图 4-9：原型地精的精灵

(5) **向躯干部位添加 Rigidbody 2D 组件**。选中全部身体部位精灵，然后在 Inspector 中单击 Add Component 按钮。在搜索框中输入 **`Rigidbody`**，添加一个 Rigidbody 2D 组件（如图 4-10 所示）。



一定要添加“Rigidbody 2D”组件，而不是“Rigidbody”。Rigidbody 组件在 3D 空间中完成模拟，这并不适合现在这个游戏。

另外，确保仅在精灵上添加 Rigidbody 2D。不要在父对象 Prototype Gnome 上添加刚体。

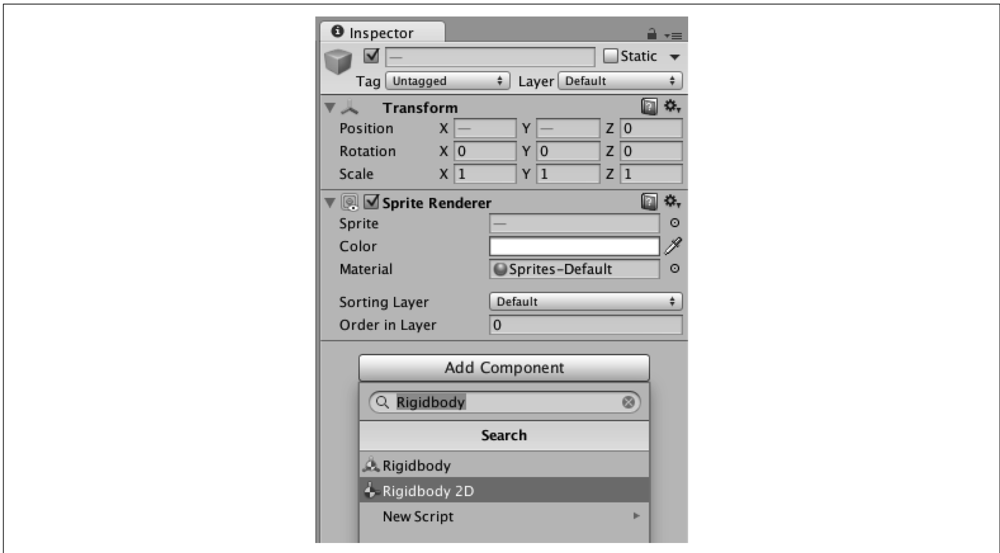


图 4-10：在精灵上添加 Rigidbody 2D 组件

(6) **在身体部位上添加碰撞器**。碰撞器定义了对象的物理形状。因为身体部位在视觉上有不同的形状，所以不同的身体部位需要不同形状的碰撞器。

a. 选择手臂和腿部精灵，添加一个 Box Collider 2D 组件。
b. 选择头部精灵，添加一个 Circle Collider 2D 组件。保留其半径不变。
c. 选择 Body 精灵，添加一个 Circle Collider 2D 组件。然后，进入该碰撞器的 Inspector，将其半径（`radius`）减小大约一半，以适应 Body 精灵。

现在就可以把地精及其身体部位链接起来了。身体部位之间的链接将通过 Hinge Joint 2D 关节完成，该关节允许对象相对于另一对象围绕中心点旋转。腿部、手臂和头部都将链接到躯干。关节的配置步骤如下所示。

(1) **选择除躯干之外的所有精灵**。躯干自身不需要任何关节，其他身体部位将通过**各自**的关节链接到躯干。
(2) **给所有选中的精灵添加 Hinge Joint 2D 组件**。通过单击 Inspector 底部的 Add Component 按钮，然后选择 Physics 2D → Hinge Joint 2D 完成。
(3) **配置关节**。在仍然选中精灵的状态下，我们将设置一个所有身体部位都具备的属性：这些部位都将连接到 Body 精灵。

将 Prototype Body 从 Hierarchy 窗格拖放到 Connected Rigid Body 框中，从而把对象链接到躯干。完成之后，铰链关节的设置应该如图 4-11 所示。

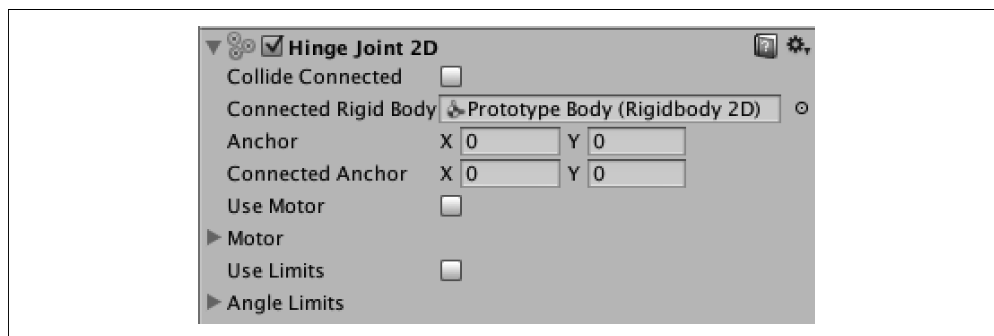


图 4-11：铰链关节的初始设置

(4) **对关节添加限制**。我们不希望对象能够旋转整圈，而想限制它们的旋转程度。这可以避免一些看上去不符合实际的行为，例如腿部从身体中穿过。

选择手臂和头部，然后选中 Use Limits。将 Lower Angle 设为 −15，Upper Angle 设为 15。接下来，选择腿部，同样选中 Use Limits。将 Lower Angle 设为 −45，Upper Angle 设为 0。
(5) **更新关节的轴点**。我们想让手臂绕肩膀旋转，让腿部绕臀部旋转。但默认情况下，关节将绕对象的中心旋转（如图 4-12 所示），这样看起来会很奇怪。

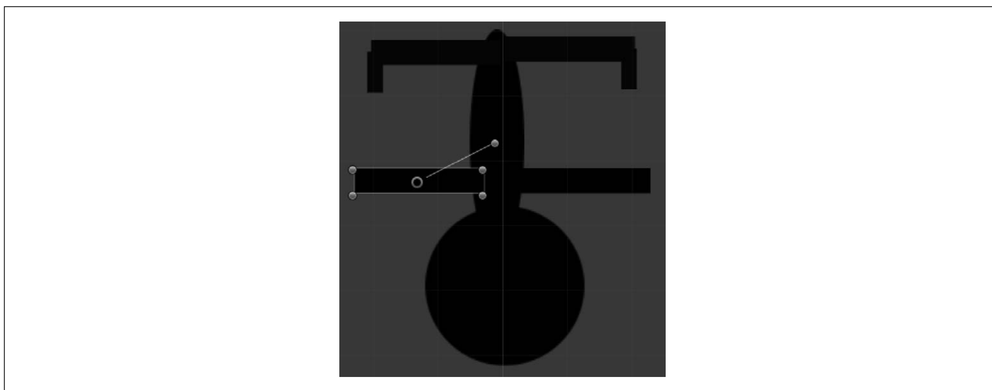


图 4-12：铰链关节的锚点的起始位置不正确

为了纠正这个问题，需要更新关节的 Anchor（锚点）以及 Connected Anchor（连接处锚点）的位置。拥有关节的身体部位将绕 Anchor 旋转，而关节**连接到**的身体部位则绕着 Connected Anchor 旋转。对于地精的关节，我们想让 Connected Anchor 和 Anchor 在同一个位置。

当选择一个有铰链关节的对象时，Anchor 和 Connected Anchor 都会显示在场景视图中：Connected Anchor 显示为一个蓝点，Anchor 显示为一个蓝色的圆圈。

选择有铰链关节的每个身体部位，将 Anchor 和 Connected Anchor 移动到正确的轴点。例如，选择右臂，将蓝点拖动到肩膀位置，这就移动了 Connected Anchor。

移动 Anchor 要稍微困难一点，因为默认情况下 Anchor 位于对象的中心，但是拖动对象的中心会导致 Unity 移动整个对象。要移动 Anchor，首先需要手动调整 Anchor 的位置：修改 Inspector 中的数字，可改变 Anchor 在场景视图中的位置。当 Anchor 不在对象的中心位置后，就可以将其拖动到正确的位置，就像拖动 Connected Anchor 一样（如图 4-13 所示）。

对双臂（连接到肩膀位置）、双腿（连接到臀部位置）和头部（连接到脖子底部）重复这个过程。

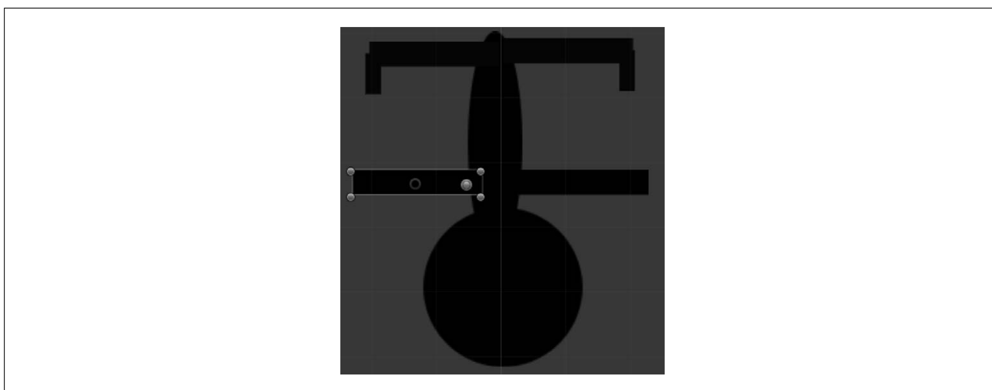


图 4-13：左臂的锚点在正确的位置；注意，点的周围有一个环，这说明 Connected Anchor 和 Anchor 在同一个位置

接下来添加连接到 Rope 对象的关节。这是一个连接到地精右腿的 Spring Joint 2D，允许绕着关节的锚点自由旋转，并限制身体离开绳索一端的距离（下一节将创建绳索）。弹簧关节的工作方式类似于现实世界中的弹簧：有弹性，可被适度拉伸。

在 Unity 中，弹簧关节主要由两个属性控制：距离和频率。距离指的是弹簧的“首选”长度，即压缩或拉伸之后，弹簧应当恢复到的长度。频率指的是弹簧的“刚度”，较低的值意味着弹簧更松。

为设置 Rope 中使用的弹簧关节，执行下面的步骤。

(1) **添加绳索关节**。选择 Prototype Leg Rope。这是右上腿部精灵。

(2) **向其添加弹簧关节**。添加一个 Spring Joint 2D。将其 Anchor（蓝色圆圈）移动到靠近腿的一端。不要移动 Connected Anchor（即移动蓝色圆圈，不移动蓝点）。图 4-14 显示了地精的锚点位置。

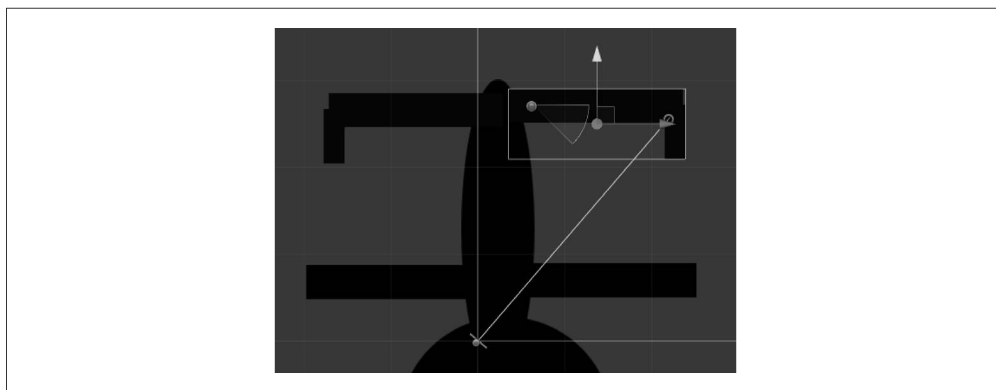


图 4-14：添加精灵关节，以将腿部连接到绳索；关节的 Anchor 靠近脚趾（另见彩插）

(3) **配置关节**。关闭 Auto Configure Distance，将关节的 Distance 设为 0.01，将 Frequency 设为 5。

(4) **运行游戏**。地精将在屏幕中部悬荡。

最后一步是缩小地精，使其以合适的大小显示在其他关卡对象旁边。

(5) **缩放地精**。选择父 Gnome 对象，将其 Scale 值的 x 和 y 值改为 0.5。这将使地精缩小一半。

现在就准备好了地精，可以添加绳索了。

4.4 绳索

绳索是游戏中第一个需要用到代码的地方，其原理如下。绳索是游戏对象的一个集合，每个游戏对象都具有刚体和弹簧关节。每个弹簧关节链接到下一个 Rope 对象，该对象又链接到下一个 Rope 对象，一直链接到绳索顶端。最上面的 Rope 对象链接到一个位置固定的刚体，从而保持自己固定在一个位置。绳索另一端连接到地精的一个组件：Rope Leg 对象。

为创建绳索，首先需要创建一个对象，作为绳索每一段的模板。然后，创建另外一个对象，它将使用此绳段对象和一些代码来生成整条绳索。执行下面的步骤来准备 Rope Segment。

(1) **创建 Rope Segment 对象**。创建一个新的空游戏对象，命名为 `Rope Segment`。
(2) **向该对象添加刚体**。添加一个 Rigidbody 2D，将其 Mass 设为 0.5，这样绳索将有一些重量感。
(3) **添加关节**。添加一个 Spring Joint 2D 组件，将其 Damping Ratio 设为 1，Frequency 设为 30。



可以自由尝试其他值。我们发现，上面设置的这些值能够让绳索比较有真实感。游戏设计的关键就在于不断调整数字。

(4) **创建一个使用此对象的预设**。打开 Assets 窗格中的 Gnome 文件夹，将 Rope Segment 对象从 Hierarchy 窗格拖放到 Assets 窗格。这将在 Assets 文件夹中创建一个新的预设文件。
(5) **删除原来的 Rope Prefab 对象**。现在已经不需要原来的对象了：我们马上编写代码来创建 Rope Segment 的多个实例，并把它们连接成整条绳索。

现在来创建 Rope 对象。

(1) **创建一个新的空游戏对象，命名为 Rope**。
(2) **改变 Rope 的图标**。因为游戏没有运行时，场景视图中是看不到绳索的，所以我们需要为绳索设置一个图标。选择新创建的 Rope 对象，然后单击 Inspector 左上角的立方体图标（如图 4-15 所示）。

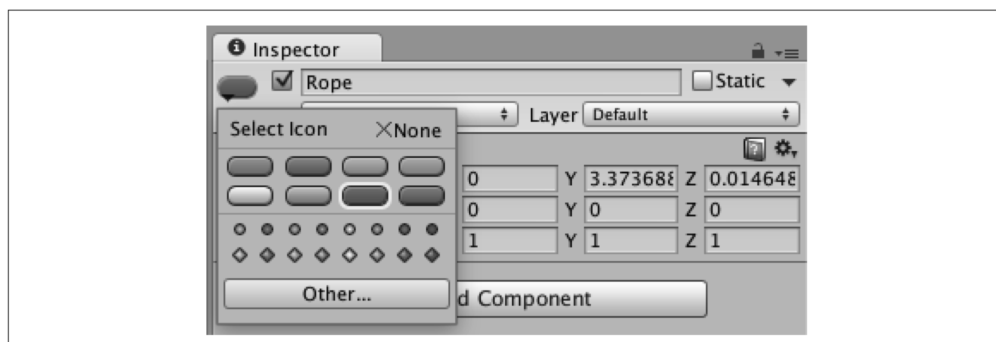


图 4-15：为 Rope 对象选择一个图标

选择红色的圆角矩形，Rope 对象将在场景中显示为一个红色的药片形状的对象（如图 4-16 所示）。

(3) **添加刚体**。单击 Add Components 按钮，向对象添加一个 Rigidbody 2D 组件。添加此刚体后，在 Inspector 中将 Body Type 改为 Kinematic。这将把对象固定在该位置，从而不会下落——这正是我们想要的效果。
(4) **添加一个线渲染器**。再次单击 Add Component 按钮，添加一个 `LineRenderer`。将新线渲染器的 Width 设为 0.075，使其具有好看的、细细的绳索外观。保留线渲染器其余设置的默认值不变。

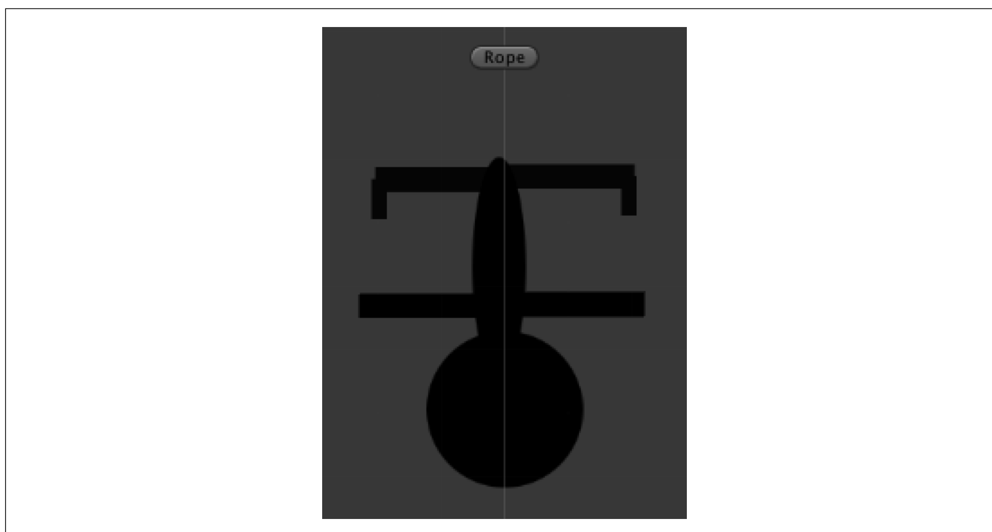


图 4-16：选择图标后，Rope 对象将出现在场景中

现在就设置好了绳索的组件，可以编写脚本来控制它们了。

4.4.1 编写控制Rope的代码

在编写代码之前，需要添加一个脚本组件。执行下面的步骤。

(1) **为 Rope 添加一个 Rope 脚本**。此脚本还不存在，但是 Unity 将为其创建一个文件。选择 Rope 对象，单击 Add Component 按钮。

输入 **Rope**；现在不会出现任何组件，因为 Unity 还没有任何名为 Rope 的组件。**能看到**的只是一个 New Script 选项（如图 4-17 所示）。选择该选项。

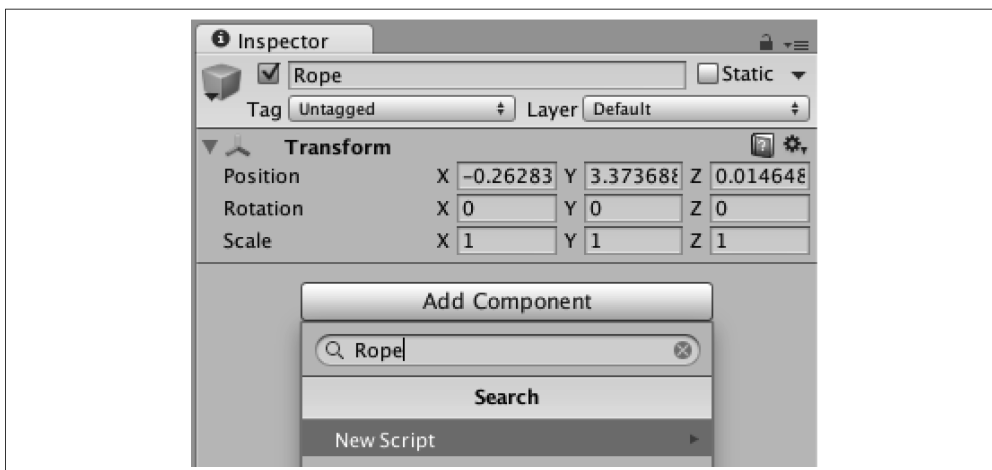


图 4-17：创建 Rope.cs 文件

Unity 会提出创建一个新的脚本文件。确保将语言设为 C Sharp，并且拼写 Rope 时使用大写的 R。单击 Create and Add 按钮。Unity 将创建 Rope.cs 文件，并将向 Rope 对象添加一个 Rope 脚本组件。

(2) **将 Rope.cs 移动到 Scripts 文件夹**。默认情况下，Unity 将把新脚本放到 Assets 文件夹中。为了保持文件整齐有序，将其移动到 Scripts 文件夹中。

(3) **在 Rope.cs 文件中添加代码**。双击打开 Rope.cs，也可以在你常用的文本编辑器中打开该文件。

在文件中添加下面的代码（稍后将介绍这段代码的作用）：

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

//连接的绳索
public class Rope : MonoBehaviour {

    //要使用的Rope Segment预设
    public GameObject ropeSegmentPrefab;

    //包含Rope Segment对象的一个List
    List<GameObject> ropeSegments = new List<GameObject>();

    //现在是在延长还是缩短绳索?
    public bool isIncreasing { get; set; }
    public bool isDecreasing { get; set; }

    //绳索末端应该关联到的rigidbody对象
    public Rigidbody2D connectedObject;

    //一段绳索的最大长度（如果需要伸长的程度超过此长度,
    //就创建一个新的绳段）
    public float maxRopeSegmentLength = 1.0f;

    //松开新绳索的速度
    public float ropeSpeed = 4.0f;

    //渲染实际绳索的LineRenderer
    LineRenderer lineRenderer;

    void Start() {

        //缓存线渲染器，这样就不需要每一帧都进行查找
        lineRenderer = GetComponent<LineRenderer>();

        //重置绳索，做好准备
        ResetLength();

    }

    //删除所有绳段，然后创建一个新的绳段
```

```
public void ResetLength() {

    foreach (GameObject segment in ropeSegments) {
        Destroy (segment);

    }

    ropeSegments = new List<GameObject>();

    isDecreasing = false;
    isIncreasing = false;

    CreateRopeSegment();
}

//将新绳段关联到绳索的顶部
void CreateRopeSegment() {

    //创建新绳段
    GameObject segment = (GameObject)Instantiate(
    ropeSegmentPrefab,
    this.transform.position,
    Quaternion.identity);

    //使绳段成为此对象的子对象，并使其保留其世界位置
    segment.transform.SetParent(this.transform, true);

    //获取绳段的刚体
    Rigidbody2D segmentBody = segment
      .GetComponent<Rigidbody2D>();

    //获取绳段的距离关节
    SpringJoint2D segmentJoint =
      segment.GetComponent<SpringJoint2D>();

    //如果绳段没有刚体或者弹簧关节，就显示错误，因为
    //二者都是我们需要的
    if (segmentBody == null || segmentJoint == null) {
    Debug.LogError("Rope segment body prefab has no " +
      "Rigidbody2D and/or SpringJoint2D!");
    return;
}

//检查后，将其添加到绳段List的开始位置
ropeSegments.Insert(0, segment);

//如果是“第一个”绳段，需要将其连接到地精

if (ropeSegments.Count == 1) {
    //将连接对象上的关节连接到绳段
    SpringJoint2D connectedObjectJoint =
      connectedObject.GetComponent<SpringJoint2D>();
```

```
            connectedObjectJoint.connectedBody
              = segmentBody;

            connectedObjectJoint.distance = 0.1f;

            //将此关节设置为最大长度
            segmentJoint.distance = maxRopeSegmentLength;
        } else {
            //这是一个额外的绳段。我们需要把顶端绳段连接到此绳段

            //获取第二个绳段
            GameObject nextSegment = ropeSegments[1];

            //获取我们需要关联到的绳段
            SpringJoint2D nextSegmentJoint =
              nextSegment.GetComponent<SpringJoint2D>();

            //连接此关节
            nextSegmentJoint.connectedBody = segmentBody;

            //使此绳段的开始位置距离前一个绳段0个
            //单位——我们将拉长前一个绳段
            segmentJoint.distance = 0.0f;
        }

        //将新绳段连接到绳索的锚点（即此对象）
        segmentJoint.connectedBody =
          this.GetComponent<Rigidbody2D>();
    }

        //缩短绳索后调用，以移除一个绳段
        void RemoveRopeSegment() {

            //如果没有两个以上的绳段，就停止移除操作
            if (ropeSegments.Count < 2) {
                return;
            }

            //获取顶端绳段及其下方的那个绳段
            GameObject topSegment = ropeSegments[0];
            GameObject nextSegment = ropeSegments[1];

            //将第二个绳段连接到绳索的锚点
            SpringJoint2D nextSegmentJoint =
              nextSegment.GetComponent<SpringJoint2D>();

            nextSegmentJoint.connectedBody =
              this.GetComponent<Rigidbody2D>();

            //删除并销毁顶端的绳段
            ropeSegments.RemoveAt(0);
```

```
        Destroy (topSegment);
    }

    //在每一帧中，根据需要增加或者减小绳索的长度
    void Update() {

        //获取顶端绳段及其关节
        GameObject topSegment = ropeSegments[0];
        SpringJoint2D topSegmentJoint =
            topSegment.GetComponent<SpringJoint2D>();

        if (isIncreasing) {

            //我们在增长绳索。如果绳段到达最大长度，就添加
            //新绳段；否则，增加顶端绳段的长度

            if (topSegmentJoint.distance >=
              maxRopeSegmentLength) {
                  CreateRopeSegment();
            } else {
                topSegmentJoint.distance += ropeSpeed *
                Time.deltaTime;
            }

        }

        if (isDecreasing) {

            //我们在缩短绳索。如果绳段长度接近于0，则删除
            //绳段；否则，减小顶端绳段的长度

            if (topSegmentJoint.distance <= 0.005f) {
                RemoveRopeSegment();
            } else {
                topSegmentJoint.distance -= ropeSpeed *
                    Time.deltaTime;
            }

        }

        if (lineRenderer != null) {
            //线渲染器根据一个点的集合绘制线。
            //必须使这些点与绳段的位置保持同步

            //线渲染器的顶点数=绳段数+加上顶端的一个点（代表绳索的锚点)+底端
            //的第一个点（代表地精）
            lineRenderer.positionCount
              = ropeSegments.Count + 2;

            //顶端的顶点总是位于绳索的位置
            lineRenderer.SetPosition(0,
              this.transform.position);
```

```
                //对于每个绳段，使对应的线渲染器顶点位于该绳段的位置
                for (int i = 0; i < ropeSegments.Count; i++) {
                    lineRenderer.SetPosition(i+1,
                          ropeSegments[i].transform.position);
                }

                //最后的顶点位于连接对象的锚点
                SpringJoint2D connectedObjectJoint =
                  connectedObject.GetComponent<SpringJoint2D>();
                lineRenderer.SetPosition(
                  ropeSegments.Count + 1,
                  connectedObject.transform.
                      TransformPoint(connectedObjectJoint.anchor)
                );
            }
        }
    }
```

代码很多，所以我们分段介绍：

```
void Start() {

    //缓存线渲染器，这样就不需要每一帧进行查找
    lineRenderer = GetComponent<LineRenderer>();

    //重置绳索，做好准备
    ResetLength();

}
```

当 Rope 对象第一次出现时，将调用其 `Start` 方法。该方法调用 `ResetLength` 方法，地精死亡时也将调用该方法。另外，设置 `lineRenderer` 变量，将其指向对象附加的线渲染器组件：

```
//删除所有绳段，然后创建一个新的绳段
public void ResetLength() {

    foreach (GameObject segment in ropeSegments) {
        Destroy (segment);
    }

    ropeSegments = new List<GameObject>();

    isDecreasing = false;
    isIncreasing = false;

    CreateRopeSegment();
}
```

`ResetLength` 方法删除所有绳段，并通过清空 `ropeSegments` 列表和 `isDecreasing/isIncreasing` 属性来重置绳索的内部状态，最后调用 `CreateRopeSegment` 来创建新的绳索：

```
//将新绳段关联到绳索的顶部
void CreateRopeSegment() {

    //创建新绳段
    GameObject segment = (GameObject)Instantiate(
        ropeSegmentPrefab,
        this.transform.position,
        Quaternion.identity);

    //使绳段成为此对象的子对象，并使其保留其世界位置
    segment.transform.SetParent(this.transform, true);

    //获取绳段的刚体
    Rigidbody2D segmentBody
      = segment.GetComponent<Rigidbody2D>();

    //获取绳段的距离关节
    SpringJoint2D segmentJoint =
        segment.GetComponent<SpringJoint2D>();

    //如果绳段没有刚体或者弹簧关节，就显示错误，因为
    //二者都是我们需要的
    if (segmentBody == null || segmentJoint == null) {
        Debug.LogError(
          "Rope segment body prefab has no " +
          "Rigidbody2D and/or SpringJoint2D!"
          );

        return;
    }

    //检查后，将其添加到绳段List的开始位置
    ropeSegments.Insert(0, segment);

    //如果是“第一个”绳段，需要将其连接到地精

    if (ropeSegments.Count == 1) {
        //将连接对象上的关节连接到绳段
        SpringJoint2D connectedObjectJoint =
            connectedObject.GetComponent<SpringJoint2D>();

        connectedObjectJoint.connectedBody =
          segmentBody;
        connectedObjectJoint.distance = 0.1f;

        //将此关节设置为最大长度
        segmentJoint.distance = maxRopeSegmentLength;
    } else {
        //这是一个额外的绳段。我们需要把顶端绳段连接到此绳段
```

```
            //获取第二个绳段
            GameObject nextSegment = ropeSegments[1];

            //获取我们需要关联到的绳段
            SpringJoint2D nextSegmentJoint =
                nextSegment.GetComponent<SpringJoint2D>();

            //连接此关节
            nextSegmentJoint.connectedBody = segmentBody;

            //使此绳段的开始位置距离前一个绳段0个单位——我们将
            //拉长前一个绳段
            segmentJoint.distance = 0.0f;
        }

        //将新绳段连接到绳索的锚点（即此对象）
        segmentJoint.connectedBody =
            this.GetComponent<Rigidbody2D>();
    }
```

`CreateRopeSegment` 创建 Rope Segment 对象的一个新副本，将其添加到绳索链的顶端。在此过程中，它将断开绳索当前的顶端（如果已经存在的话），将其重新连接到新创建的绳段。然后，代码将新绳段连接到 Rope 对象自身所附加的 `Rigidbody2D` 组件。

如果新绳段是目前为止创建的唯一绳段，则该绳段把自身附加到 `connectedObject` 刚体。此变量将被设为地精的腿部：

```
    //缩短绳索后调用，以移除一个绳段
    void RemoveRopeSegment() {

        //如果没有两个以上的绳段，就停止移除操作
        if (ropeSegments.Count < 2) {
            return;
        }

        //获取顶端绳段及其下方的那个绳段
        GameObject topSegment = ropeSegments[0];
        GameObject nextSegment = ropeSegments[1];

        //将第二个绳段连接到绳索的锚点
        SpringJoint2D nextSegmentJoint =
            nextSegment.GetComponent<SpringJoint2D>();

        nextSegmentJoint.connectedBody =
            this.GetComponent<Rigidbody2D>();

        //删除并销毁顶端的绳段
        ropeSegments.RemoveAt(0);
        Destroy (topSegment);

    }
```

RemoveRopeSegment 的作用相反。删除顶部绳段，将下面的绳段连接到 Rope 刚体。注意，如果只有一个绳段，则 RemoveRopeSegment 什么也不做。也就是说，就算绳索一直收缩，也不会完全消失：

```
//在每一帧中，根据需要增加或者减小绳索的长度
void Update() {

    //获取顶端绳段及其关节
    GameObject topSegment = ropeSegments[0];
    SpringJoint2D topSegmentJoint =
        topSegment.GetComponent<SpringJoint2D>();

    if (isIncreasing) {

        //我们在增长绳索。如果绳段到达最大长度，就添加
        //新绳段；否则，增加顶端绳段的长度

        if (topSegmentJoint.distance >=
            maxRopeSegmentLength) {

            CreateRopeSegment();

        } else {

            topSegmentJoint.distance += ropeSpeed *
                Time.deltaTime;

        }

    }

    if (isDecreasing) {

        //我们在缩短绳索。如果绳段长度接近于0，则删除
        //绳段；否则，减小顶端绳段的长度

        if (topSegmentJoint.distance <= 0.005f) {
            RemoveRopeSegment();
        } else {
            topSegmentJoint.distance -= ropeSpeed *
                Time.deltaTime;
        }
    }

    if (lineRenderer != null) {
        //线渲染器根据一个点的集合绘制线。必须使这些点与绳段的位置保持同步

        //线渲染器的顶点数=绳段数+加上顶端的一个点（代表绳索的锚点）+底端
        //的第一个点（代表地精）
        lineRenderer.positionCount =
          ropeSegments.Count + 2;
```

```
            //顶端的顶点总是位于绳索的位置
            lineRenderer.SetPosition(0,
              this.transform.position);

            //对于每个绳段，使对应的线渲染器顶点位于该绳段的位置
            for (int i = 0; i < ropeSegments.Count; i++) {
                lineRenderer.SetPosition(
                    i+1,
                    ropeSegments[i].transform.position
                );
            }

            //最后的顶点位于连接对象的锚点
            SpringJoint2D connectedObjectJoint =
              connectedObject.GetComponent<SpringJoint2D>();

            var lastPosition = connectedObject
                .transform
                .TransformPoint(
                    connectedObjectJoint.anchor
                );

            lineRenderer.SetPosition(
              ropeSegments.Count + 1,
              position
            );
        }
    }
```

每次调用 `Update` 方法时（即每当游戏重绘屏幕时），绳索会检查是 `isIncreasing` 还是 `isDecreasing` 为 `true`。

如果发现 `isIncreasing` 为 `true`，那么绳索将逐渐增加顶端绳段的弹簧关节的 `distance` 属性。如果此属性大于等于 `maxRopeSegment` 变量，将创建新的绳段。

反之，如果 `isDecreasing` 为 `true`，那么将减小 `distance` 属性。如果此值接近 `0`，则移除顶端的绳段。

最后，更新 `LineRenderer`，使定义线条显示位置的顶点与绳段对象的位置相符合。

4.4.2 配置绳索

设置好 Rope 的代码后，现在可以让场景中的对象使用这些代码了。为此，执行下面的步骤。

(1) **配置 Rope 对象**。选择 Rope 游戏对象。将 Rope Segment 预设拖动到绳索的 Rope Segment Prefab 框中，将地精的 Rope Leg 对象拖动到绳索的 Connected Object 框中。保留其他设置的默认值，即 Rope.cs 文件中定义的值。完成上述操作后，Rope 的 Inspector 如图 4-18 所示。

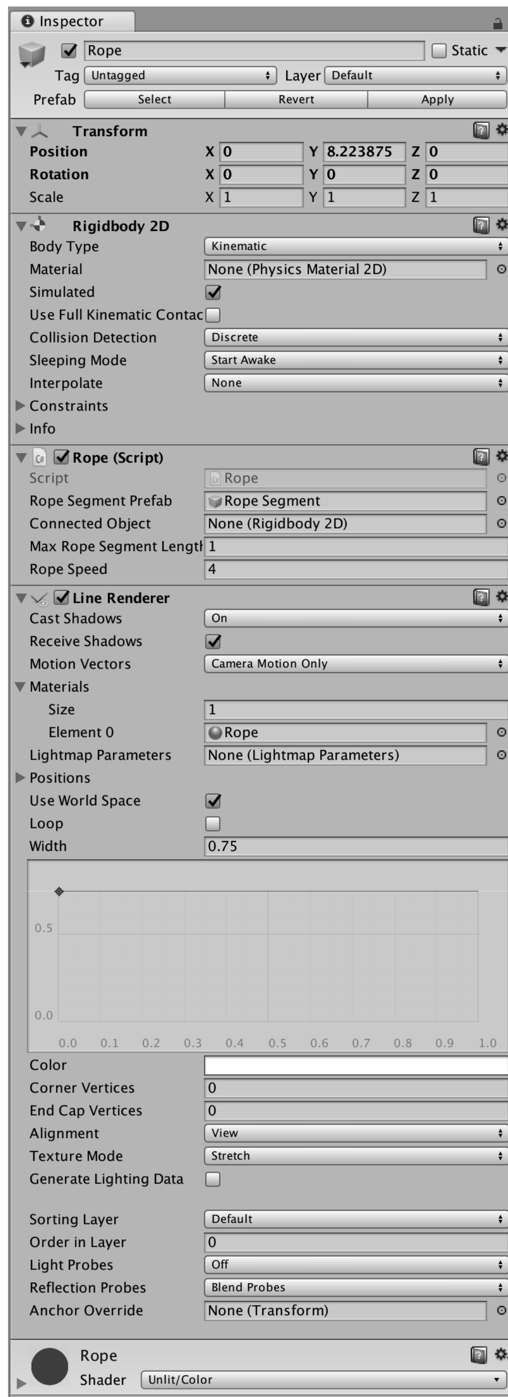


图 4-18：配置好的 Rope 对象

(2) **运行游戏**。现在地精将挂在 Rope 对象上悬荡，并且我们能够看到绳索连接到地精稍上方的一个位置。

对于 Rope 对象，还剩下一个步骤：设置 Line Renderer 使用的材质。

(1) **创建材质**。打开 Assets 菜单，选择 Create → Material。将新材质命名为 Rope。
(2) **设置 Rope 材质**。选择新的 Rope 材质，在 Inspector 中打开 Shader 菜单。选择 Unlit → Color。Inspector 将显示新着色器的参数，即一个颜色框。单击颜色框，从弹出的窗口中选择深棕色，以修改材质的颜色。
(3) **让 Rope 使用新材质**。选择 Rope 对象，打开 Materials 属性。将刚才创建的 Rope 材质拖放到 Element 0 框中。
(4) **再次运行游戏**。现在绳索显示为棕色。

4.5 小结

现在，游戏的基本结构已开始成形。游戏最重要的两个部分已经能够工作了：布娃娃地精，以及悬挂地精的绳索。

第 5 章将创建一些系统来使用这些对象实现游戏的玩法，会很有趣的！

第5章

建立游戏玩法

现在已经创建了地精和绳索对象，是时候建立一个系统来让用户为游戏提供输入了。

这个过程分为两个部分：首先，添加一个脚本，使得倾斜手机时，地精随之摇晃；然后，添加延长和缩短绳索的按钮。

完成这些设置后，我们将实现驱动游戏本身的代码：首先完成地精最终将会使用的许多设置工作，然后实现一个管理器对象，用来跟踪一些重要的游戏状态。

5.1 输入

现在，我们已经到了需要从设备获取输入的时候，因此必须确保 Unity Editor 能够接收输入。否则，测试游戏的唯一方法是构建游戏，并将其安装到一个设备上，需要的时间会比较多。Unity 旨在帮助我们快速测试修改，如果要等待游戏构建完成，就会大大降低进度。

5.1.1 Unity Remote

为了允许快速向 Unity Editor 提供输入，Unity 在 App Store 中提供了一个叫作 Unity Remote 的应用。Unity Remote 通过手机数据线连接到 Unity Editor，在 Unity Editor 中玩游戏时，手机会显示游戏窗口中内容的副本，并将所有触摸和传感器信息发回脚本。这样，不必构建游戏就可以进行测试。只要在手机上启动 Unity Remote，然后就像已经安装了游戏那样玩游戏即可。

Unity Remote 有如下几个缺点。

- 为了在手机上显示游戏，Unity 会严重压缩图片。除了图片显示质量下降以外，把图片传输到手机还会增加延迟，降低帧率。

- 因为游戏在计算机上运行，所以帧率不会与在手机上运行时一样。如果场景需要渲染大量图形，或者脚本在每一帧的运行时间都很久，那么性能不会与在手机上运行时一样。
- 最后，只有把手机连接到计算机时，Unity Remote 才会工作。

使用 Unity Remote，首先需要从设备的应用商店中下载，然后启动它，并使用 USB 数据线把手机连接到计算机。之后单击 Play 按钮。游戏将出现在设备上。

如果在设备上看不到任何东西，则打开 Edit 菜单，选择 Project Settings → Editor。Editor 设置将在 Inspector 中打开。将 Device 设置改为手机。



关于在设备上安装 Unity Remote 的最新说明，请查阅 Unity 的文档（https://docs.unity3d.com/Manual/UnityRemote5.html）。

5.1.2　添加倾斜控制

此功能由两个脚本驱动：`InputManager`（读取加速计信息）和 `Swinging`（读取 `InputManager` 的输入，并向刚体应用一个横向力——这里的刚体是地精的躯干）。

1. 创建单例类

`InputManager` 是一个单例（singleton）对象。这意味着在场景中，只有一个 `InputManager`，其他所有对象将访问该 `InputManager`。因为后面将在代码中添加其他类型的单例，所以创建一个让代码多个部分能够重用的类也很合理。为了准备 `InputManager` 使用的 `Singleton` 类，执行下面的步骤。

(1) **创建 Singleton 脚本**。打开 Assets 菜单，选择 Create → C# Script，在 Scripts 文件夹中创建一个新的 C# 脚本资源。将该脚本命名为 Singleton。

(2) **添加 Singleton 代码**。打开 Singleton.cs，将其内容替换为下面的代码：

```
using UnityEngine;
using System.Collections;

//此类允许其他对象引用单个共享对象
//GameManager和InputManager使用此类

//为使用此类，需要进行继承，如：
//public class MyManager : Singleton<MyManager> {  }

//然后就可以访问此类的单个共享实例，如：
//MyManager.instance.DoSomething();

public class Singleton<T> : MonoBehaviour
  where T : MonoBehaviour {

  //此类的单个实例
  private static T _instance;

  //访问器。第一次调用时，将设置_instance
  //如果找不到合适的对象，将记录一个错误
```

```
    public static T instance {
      get {
        //如果还没有设置_instance……
        if (_instance == null)
        {
          //尝试找到该对象
          _instance = FindObjectOfType<T>();

          //如果找不到，就记录错误
          if (_instance == null) {
            Debug.LogError("Can't find " +
              typeof(T) + "!");
          }
        }

        //返回实例供使用
        return _instance;
      }
    }
  }
```

`Singleton` 类的工作方式如下：其他类将是这个模板类的子类，并获得一个名为 `instance` 的静态属性。此属性始终指向该类的共享实例。这意味着当其他脚本请求 `InputManager.instance` 时，总是会得到单例 `InputManager`。

这种方法的优点是，需要 `InputManager` 的脚本不需要连接到 `InputManager` 的变量。

2. 实现InputManager单例

创建了 `Singleton` 类以后，接下来创建 `InputManager`。

(1) **创建 `InputManager` 游戏对象**。创建一个新游戏对象，命名为 `InputManager`。

(2) **创建并添加 `InputManager` 脚本**。选择 `InputManager` 对象，单击 Add Component 按钮。输入 `InputManager`，然后选择创建一个新脚本。确保脚本的名称是 `InputManager`，语言是 C#。

(3) **在 InputManager.cs 中添加代码**。打开刚刚创建的 InputManager.cs 文件，在其中添加下面的代码：

```
using UnityEngine;
using System.Collections;

//将加速计数据转换为侧向运动信息
public class InputManager : Singleton<InputManager> {

  //移动程度。-1.0=向左到头，+1.0=向右到头
  private float _sidewaysMotion = 0.0f;

  //将此属性声明为只读属性，使其他类不能改变此属性
  public float sidewaysMotion {
    get {
      return _sidewaysMotion;
    }
  }
```

```
  //在每一帧中存储倾斜度
  void Update () {
    Vector3 accel = Input.acceleration;

    _sidewaysMotion = accel.x;
  }
}
```

在每一帧中，InputManager 类通过内置的 Input 类，从加速计采样数据，并将数据的 X 分量（测量施加到设备左右两侧力的大小）存储到一个变量中。使用公共只读属性 sidewaysMotion 可访问此变量。



只读属性用于防止其他类误写该值。

简言之，如果其他任何类想要知道手机在左右方向上倾斜多少，只需要请求 InputManager.instance.sidewaysMotion 即可。

现在来编写 Swinging 代码。

(1) **选择地精的 Body 对象。**

(2) **创建并添加新的 C# 脚本，命名为 Swinging.cs。**在脚本中添加下面的代码：

```
using UnityEngine;
using System.Collections;

//用InputManager向对象应用侧向力，使地精左右晃动
public class Swinging : MonoBehaviour {

  //晃动程度如何？数字越大，晃动程度越大
  public float swingSensitivity = 100.0f;

  //使用FixedUpdate，而不是Update，以更适合物理引擎
  void FixedUpdate() {

    //如果没有（或不再有）刚体，就删除此组件
    if (GetComponent<Rigidbody2D>() == null) {
      Destroy (this);
      return;
    }

    //从InputManager获取倾斜量
    float swing = InputManager.instance.sidewaysMotion;

    //计算要应用的力
    Vector2 force =
      new Vector2(swing * swingSensitivity, 0);

    //应用力
    GetComponent<Rigidbody2D>().AddForce(force);
  }

}
```

每次物理系统更新时，`Swinging` 类都会运行代码。首先，它检查对象是否仍有 `Rigidbody2D` 组件。如果没有，就立即返回。如果有，就从 `InputManager` 获取 `sidewaysMotion`，用来创建一个 `Vector2`，将其作为力应用到对象的刚体。

(3) **运行游戏**。在手机上启动 Unity Remote，左右倾斜手机，地精将随之左右移动。



如果转动手机的程度太大，Unity Remote 可能切换到水平模式，使画面拉伸。为了防止这种情况出现，可以关闭设备的屏幕旋转功能。

5.1.3　控制绳索

现在，使用 Unity GUI 按钮来添加延长和缩短绳索的按钮。用户按下 Down 按钮时，通知 Rope 开始延长；用户松开按钮时，Rope 将停止延长。Up 按钮的工作方式类似，使 Rope 开始和停止缩短。

(1) **添加按钮**。打开 GameObject 菜单，选择 UI → Button。这将添加按钮、显示按钮的 Canvas，以及处理按钮输入的 EventSystem。（现在还不需要担心这些对象。）将按钮的游戏对象命名为 Down。

(2) **将按钮放到右下角**。选择 Down 按钮，单击 Inspector 左上角的 Anchor 按钮。按住 Shift 键和 Alt 键（Mac 上为 Option 键），并单击 bottom-right 选项（如图 5-1 所示）。此操作把按钮的锚点和位置设为右下角。于是，执行此操作后，按钮将移动到屏幕的右下角。

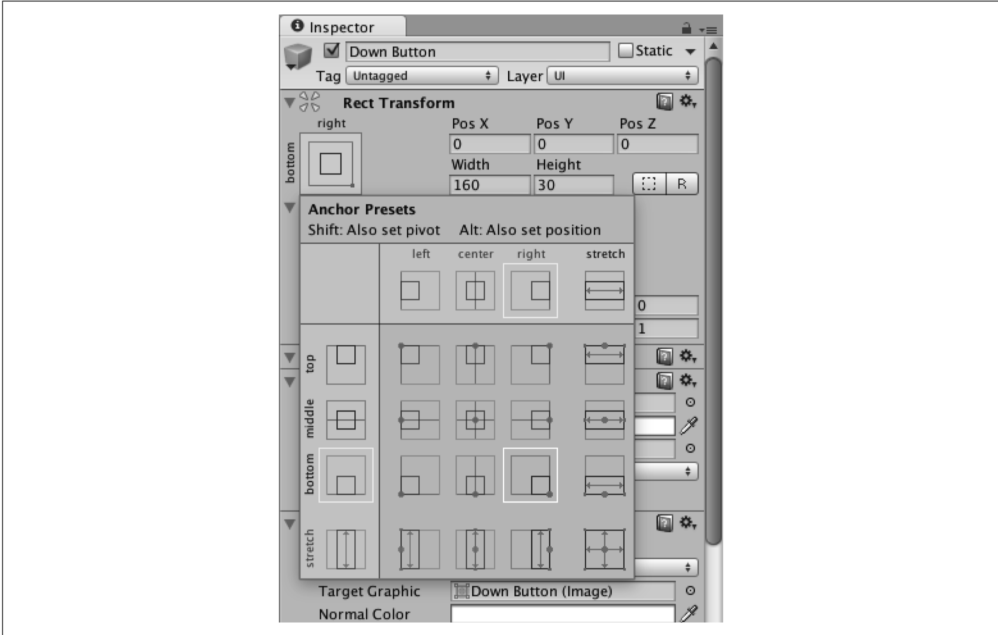


图 5-1：将 Down 按钮的锚点设为右下角；在此屏幕截图中，按下了 Shift 键和 Alt 键，这意味着单击 bottom-right 锚点会设置锚点及按钮的位置

(3) **将按钮的文本设置为 Down**。Button 只有一个名为 Text 的子对象，该对象是包含在按钮内的标签。选择该对象，在 Inspector 中找到其所附加的 Text 组件，将 Text 属性改为 Down。按钮上的文字将显示为 Down。

(4) **从 Button 对象中移除 Button 组件**。单击 Button 组件右上角的齿轮图标，选择 Remove Component。



可能出乎你的意料，我们实际上并不想让这个 UI 元素的行为与“普通的”按钮一样。

普通的按钮在被“单击”时——用户把一根手指放到按钮上，然后松开手指——会发送一个事件。只有手指松开时才会发送事件，这不能满足我们的需求。我们需要的是，手指放到按钮上时发送一个事件，手指从按钮上松开时发送另一个事件。

因此，我们将手动添加向绳索发送消息的组件。

(5) **向 Button 对象添加 Event Trigger 组件**。该组件负责监视交互，当发生交互时发送消息。

(6) **添加 Pointer Down 事件**。单击 Add New Event Type 按钮，从显示的列表中选择 Pointer Down。

(7) **将 Rope 的 `isIncreasing` 属性连接到事件**。单击 Pointer Down 列表中的 + 按钮，将出现一个新条目，如图 5-2 所示。

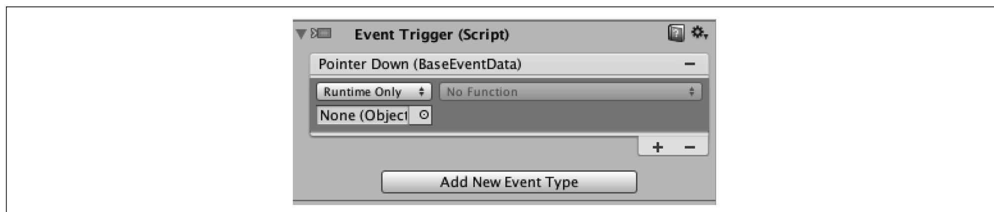


图 5-2：列表中的新事件

将 Rope 对象从 Hierarchy 窗格拖动到出现的对象框中。

将 Function 从 No Function 改为 Rope → isIncreasing（选择此选项时，下拉菜单将显示 Rope.isIncreasing）。这样，当手指放到按钮上时，按钮将修改绳索的 `isIncreasing` 属性。

将出现的复选框从未选中改为选中状态，这将把 `isIncreasing` 属性改为 `true`。

完成上述操作后，Pointer Down 事件中的新条目将如图 5-3 所示。

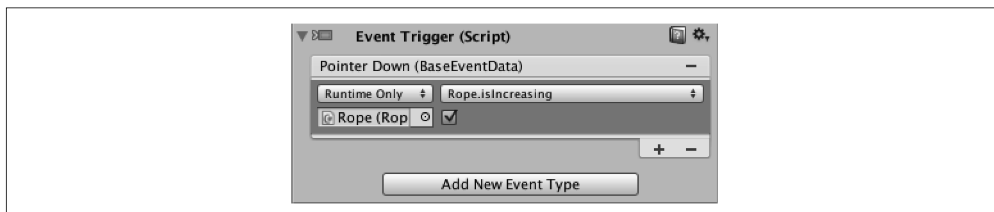


图 5-3：配置好的 Pointer Down 事件

(8) **添加一个 Pointer Up 事件，将 Rope 的 `isIncreasing` 属性设为 `false`**。将手指从按钮上松开时，我们想让绳索停止延长。

单击 Add New Event Type 按钮，在 Event Trigger 中添加新事件 Pointer Up，并取消选中 Rope 的 `isIncreasing` 属性的复选框。这样，当手指松开时，`isIncreasing` 属性将变为 `false`。

完成上述操作后，Event Trigger 的 Inspector 应如图 5-4 所示。

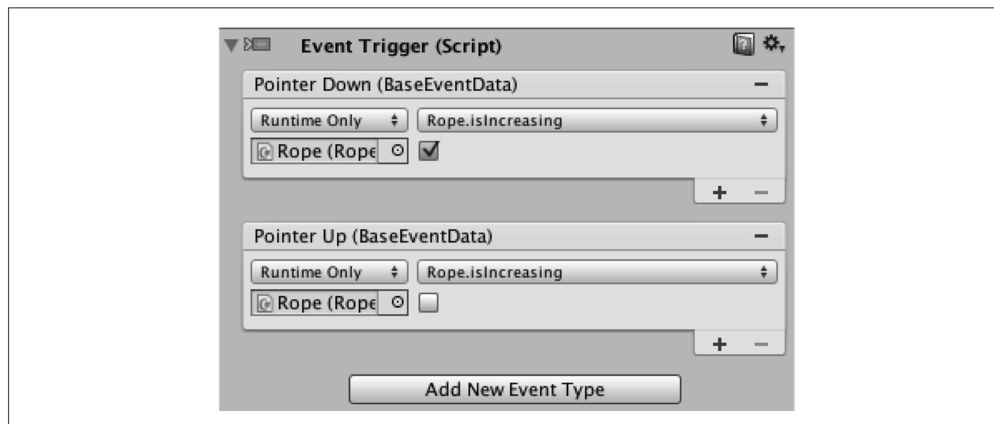


图 5-4：完全配置好的 Down 按钮的 Event Trigger

(9) **测试 Down 按钮**。开始游戏，单击并按住 Down 按钮。绳索应该开始延长，当松开鼠标左键时，鼠标应该停止延长。如果没有，则需要检查为 Down 按钮配置的事件：Pointer Down 应该将 `isIncreasing` 设为 `true`，Pointer Up 应该将 `isIncreasing` 设为 `false`。

(10) **添加 Up 按钮**。现在为缩短绳索的按钮重复相同的过程。添加一个新按钮，像 Down 按钮一样将其放到右下角，然后向上稍微移动一些。

将新按钮的标签改为 Up，删除 Button 组件，并添加一个 Event Trigger（有两个事件类型：Pointer Down 和 Pointer Up）。让这两个 Event Trigger 修改 Rope 的 `isDecreasing` 属性。

两个按钮仅有的区别在于标签文本以及影响的属性。除此之外，二者完全相同。

(11) **测试 Up 按钮**。再次玩游戏。现在应该能够延长和缩短绳索了。

在改变绳索长度的同时，还可以使用在手机上运行的 Unity Remote 左右晃动地精。

祝贺你：输入系统的核心部分已经完成了！

5.1.4　使摄像机跟随地精

现在，如果按下 Down 按钮，绳索将坠下地精，直到屏幕中看不到它。我们需要让摄像机跟随地精。

为此，我们将创建一个附加到 Camera 的脚本，将 Camera 的 y 坐标（即垂直位置）对应到另一个对象的 y 坐标。将另外一个对象配置为地精后，摄像机将跟随地精。此脚本将附加到摄像机，并将配置为跟踪地精的躯干。创建脚本的步骤如下。

(1) **添加 `CameraFollow` 脚本**。在 Hierarchy 中选择 Camera，并添加一个新的 C# 组件，命名为 Camera Follow。

(2) 在 CameraFollow.cs 中添加下面的代码：

```
// 调整摄像机，使其在特定的限定值内，始终对齐目标对象的y位置
public class CameraFollow : MonoBehaviour {

  //我们想要与此对象的y位置对齐
  public Transform target;

  //摄像机能够到达的最高点
  public float topLimit = 10.0f;

  //摄像机能够到达的最低点
  public float bottomLimit = -10.0f;

  //朝向目标移动的速度
  public float followSpeed = 0.5f;

  //全部对象的位置都更新后，确定此摄像机应该在什么位置
  void LateUpdate () {

    //如果有一个目标……
    if (target != null) {

      //就获取其位置
      Vector3 newPosition = this.transform.position;

      //计算出摄像机应该在什么位置
      newPosition.y = Mathf.Lerp (newPosition.y,
        target.position.y, followSpeed);

      //使新位置位于限定值以内
      newPosition.y =
        Mathf.Min(newPosition.y, topLimit);
      newPosition.y =
        Mathf.Max(newPosition.y, bottomLimit);

      //更新位置
      transform.position = newPosition;
    }

  }

  //在编辑器中选中时，绘制一条从顶端限定值到底端限定值的线条
  void OnDrawGizmosSelected() {
    Gizmos.color = Color.yellow;

    Vector3 topPoint =
      new Vector3(this.transform.position.x,
        topLimit, this.transform.position.z);
    Vector3 bottomPoint =
```

```
        new Vector3(this.transform.position.x,
          bottomLimit, this.transform.position.z);

      Gizmos.DrawLine(topPoint, bottomPoint);
    }
  }
```

CameraFollow 代码使用 LateUpdate 方法，该方法在其他所有对象运行完各自的 Update 方法之后运行。Update 常用于更新对象的位置，也就是说，使用 LateUpdate 方法意味着代码将在位置更新完成**之后**运行。

CameraFollow 跟随其所附加到的对象变换的 *y* 位置，同时确保该位置不会高过或者低于特定的阈值。这意味着当绳索收缩到最短时，摄像机不会显示井口上方的空白区域。另外，代码使用了 Mathf.Lerp 函数来计算接近目标的一个位置。这样一来，摄像机就会“大致”跟随对象——followSpeed 参数的值越接近 1，摄像机的移动速度就越快。

为了在视觉上表现阈值，我们实现了 OnDrawGizmosSelected 方法。每当选择摄像机的时候，Unity Editor 自己会使用该方法，绘制一条从高阈值到低阈值的线。如果使用 Inspector 修改 topLimit 和 bottomLimit 属性，就会看到这条线的长度发生变化。

(3) **配置** Camera Follow **组件**。将地精的 Body 对象拖动到 Target 框中（如图 5-5 所示），保留其他属性不变。

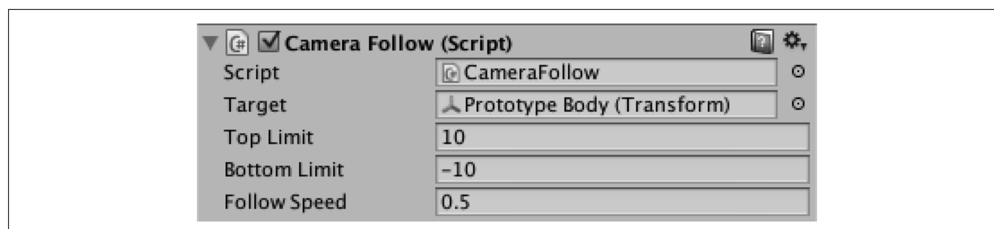


图 5-5：设置 CameraFollow 脚本

(4) **测试摄像机**。运行游戏，使用 Down 按钮降下地精。摄像机将跟随地精。

5.1.5 脚本与调试

从现在开始，代码将变得更加复杂，所以现在我们先来讨论如何找出和修复脚本中的问题。

有时候，由于拼写错误或者逻辑错误，脚本的行为并不符合预期。可以使用 MonoDevelop 中的调试功能来跟踪和解决脚本中存在的问题。你可以在代码中设置断点，检查程序的状态，以及精确控制脚本的执行。



虽然可以使用任何文本编辑器来编辑脚本，但是在开发时，需要使用专用的开发环境；也就是说，要使用 MonoDevelop 或者 Visual Studio。本书使用的是 MonoDevelop，如果你想使用 Visual Studio，可以参考 Microsoft 提供的优秀文档（https://msdn.microsoft.com/en-us/library/k0k771bt.aspx）。

设置断点

为了探索此功能，我们将在刚才编写的 Rope 脚本中设置一个断点，以便更细致地观察脚本的行为。为此，执行下面的步骤。

(1) **在 MonoDevelop 中打开 Rope.cs。**

(2) **找到 `Update` 方法。**具体来说，找到下面的代码行：

```
if (topSegmentJoint.distance >= maxRopeSegmentLength) {
```

(3) **在该代码行所在的灰色条的左侧单击。**这将添加一个断点，如图 5-6 所示。

```
// Every frame, increase or decrease the rope's length if neccessary
void Update() {

    // Get the top segment and its joint.
    GameObject topSegment = ropeSegments[0];
    SpringJoint2D topSegmentJoint =
        topSegment.GetComponent<SpringJoint2D>();

    if (isIncreasing) {

        // We're increasing the rope. If it's at max length,
        // add  a new segment; otherwise, increase the top
        // rope segment's length.

        if (topSegmentJoint.distance >= maxRopeSegmentLength) {
            CreateRopeSegment();
        } else {
            topSegmentJoint.distance += ropeSpeed *
                Time.deltaTime;
        }

    }
```

图 5-6：添加断点

接下来，我们将把 MonoDevelop 连接到 Unity。这意味着当到达断点时，MonoDevelop 将进行干预，暂停 Unity。

(4) **在 MonoDevelop 中单击窗口左上角的 Play 按钮，如图 5-7 所示。**

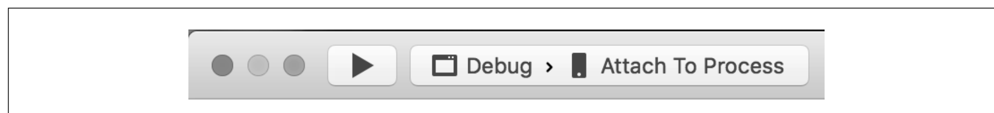


图 5-7：MonoDevelop 窗口左上角的 Play 按钮

(5) **在显示的窗口中，单击 Attach 按钮，如图 5-8 所示。**

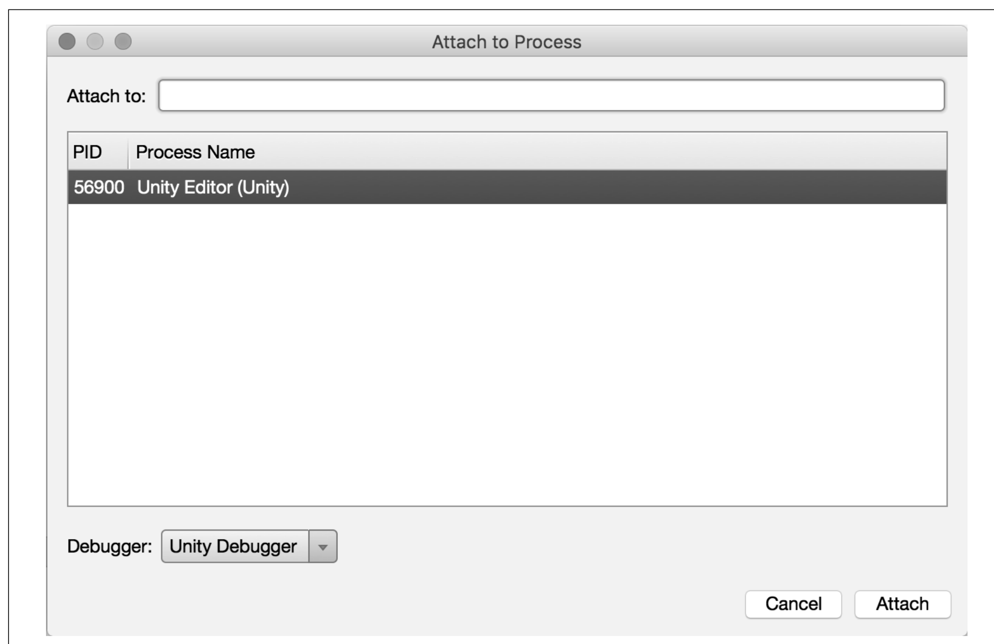


图 5-8：Attach to Process 窗口

现在 MonoDevelop 就连接到了 Unity。当到达断点时，Unity 会暂停，允许你调试代码。



当提到“Unity 会暂停”时，不是说 Unity 内的**游戏**会暂停，就像单击 Pause 按钮时那样。相反，**整个 Unity 应用程序**将会冻结，在你告诉 MonoDevelop 继续运行之前不会继续执行。如果看起来 Unity 挂起了，也不要担心。

(6) **运行游戏，并单击 Down 按钮。**

执行此操作后，Unity 将会冻结，MonoDevelop 将会出现。设置了断点的代码行将会高亮显示，指出这是当前的执行点。

现在，我们可以更深入地观察程序的状态。在编辑器底部，界面分成了两个窗格：Locals 窗格和 Immediate 窗格。（由于具体运行环境不同，可能打开的选项卡也不同。如果是这样，可以单击选项卡打开对应的窗格。）

Locals 窗格用于查看当前在作用域内的变量列表。

(7) **在 Locals 窗格中打开 `topSegmentJoint` 变量。**这将显示该变量内的字段列表，使你能够查看它们，如图 5-9 所示。

Name	Value	Type
▶ this	{Rope (Rope)}	Rope
▶ topSegment	{Rope Segment(Clone) (UnityEngine.GameObject)}	UnityEngine.GameOb
▼ topSegmentJoint	{Rope Segment(Clone) (UnityEngine.SpringJoint2D)}	UnityEngine.SpringJo
▶ base	{UnityEngine.AnchoredJoint2D}	UnityEngine.Anchorec
autoConfigureDistance	false	bool
dampingRatio	1	float
distance	1	float
frequency	30	float

图 5-9：Locals 窗格，显示了 `topSegmentJoint` 内的数据



在 Immediate 窗格中，可以键入 C# 代码来查看其结果。例如，通过键入 **`topSegmentJoint.distance`**，也可以访问图 5-9 中 `topSegmentJoint` 的 `distance` 属性信息。

调试完代码后，需要告诉调试器让 Unity 继续运行。有两种方法：断开调试器，或者保持连接调试器，并发出继续执行的信号。

如果断开调试器，将不再遇到断点，再次调试时需要重新连接调试器。如果保持连接调试器，那么下一个遇到的断点将再次暂停游戏。

- 要断开调试器，单击 Stop 按钮，如图 5-10 所示。

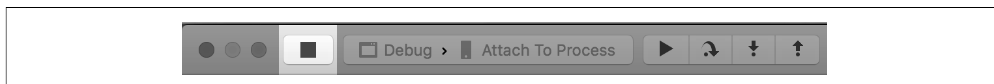


图 5-10：停止调试器

- 要保持连接调试器并继续执行，单击 Continue 按钮，如图 5-11 所示。

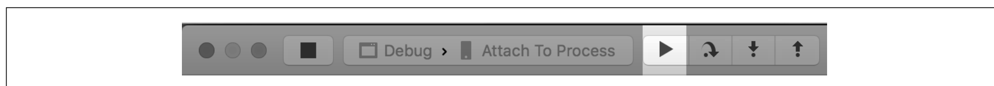


图 5-11：继续执行

5.2　创建地精的代码

现在是时候完成地精的设置了。地精需要知道自己在游戏中的状态，以及发生在自己身上的事情。

具体来说，我们想让地精表现出如下行为。

- 当受到伤害时，应根据受到的伤害类型显示对应的粒子效果。
- 死亡时，应该展示以下效果。
  - 根据伤害类型，更新不同身体部位的精灵，并断开某些部位。
  - 在死亡后应该很快创建一个上升的 Ghost 对象。
  - 当肢体断开时，应该从躯干喷出血流；我们需要知道对于每个肢体，血在什么地方喷射。
  - 当断开的肢体停止移动时，应该丢失所有物理效果，使其不会影响玩家（我们不希望死去的地精在井底堆积起来，使玩家无法拾取宝藏）。
- 应该跟踪自己是否抱着宝藏；当抱着时，应该把手臂的精灵换为显示地精抱着宝藏的精灵。
- 应该存储一些重要的信息，例如摄像机应该跟随什么对象，绳索应该附加到什么刚体。
- 应该跟踪自己是否死亡。

记住，地精是独立于整个游戏的状态的。地精不会跟踪玩家是否获胜，只会管理自身的状态。我们最终还会创建一个对象来管理整个游戏的状态，并让地精死亡。

为了实现这个系统，我们需要一个脚本来将地精作为一个整体进行管理。另外，还需要给每个身体部位添加一个脚本（管理它们的精灵，以及在它们断开后停止物理计算）。

还需要添加一些额外的信息来跟踪喷血的位置。这些位置由游戏对象表示（因为它们可以放在场景中），每个身体部位都将引用对应的“喷血”位置。

我们首先创建身体部位脚本，然后创建地精的脚本。这个顺序是因为主地精脚本需要知道身体部位脚本，而身体部位脚本不需要知道主地精脚本。

(1) **创建 BodyPart.cs 文件**。创建一个新的 C# 脚本，命名为 BodyPart.cs。在其中添加下面的代码：

```
[RequireComponent (typeof(SpriteRenderer))]
public class BodyPart : MonoBehaviour {

  //当调用ApplyDamageSprite，并将伤害类型设为slicing时使用的精灵
  public Sprite detachedSprite;

  //当调用ApplyDamageSprite，并将伤害类型设为burning时调用的精灵
  public Sprite burnedSprite;

  //代表血流在躯干上出现的位置和旋转
  public Transform bloodFountainOrigin;

  //如果为true，此对象将在停止移动时移除其碰撞、关节和刚体
  bool detached = false;

  //将此对象与其父对象解除关联，并将其标记为需要删除其物理行为
  public void Detach() {
    detached = true;
```

```
    this.tag = "Untagged";

    transform.SetParent(null, true);
  }

  //在每一帧中，如果肢体断开，并且刚体处于休眠状态，
  //就删除物理行为。这意味着断开的肢体不会影响到
  //地精的运动
  public void Update() {

    //如果肢体没有断开，就不做处理
    if (detached == false) {
      return;
    }

    //刚体在休眠吗？
    var rigidbody = GetComponent<Rigidbody2D>();

    if (rigidbody.IsSleeping()) {

      //如果是，就删除其全部关节……
      foreach (Joint2D joint in
        GetComponentsInChildren<Joint2D>()) {
        Destroy (joint);
      }

      //……刚体……
      foreach (Rigidbody2D body in
        GetComponentsInChildren<Rigidbody2D>()) {
        Destroy (body);
      }

      //……及碰撞器
      foreach (Collider2D collider in
        GetComponentsInChildren<Collider2D>()) {
        Destroy (collider);
      }

      //最后，删除此脚本
      Destroy (this);
    }
  }

  //根据受到的伤害类型，替换掉此身体部位的精灵
  public void ApplyDamageSprite(
    Gnome.DamageType damageType) {

    Sprite spriteToUse = null;

    switch (damageType) {

    case Gnome.DamageType.Burning:
      spriteToUse = burnedSprite;
      break;
```

```
      case Gnome.DamageType.Slicing:
        spriteToUse = detachedSprite;

        break;
      }

      if (spriteToUse != null) {
        GetComponent<SpriteRenderer>().sprite =
          spriteToUse;
      }

    }

  }
```



这段代码还不能通过编译，因为用到了我们还没有编写的 Gnome.DamageType 类型。稍后编写 Gnome 类的时候，我们将添加这个类型。

BodyPart 脚本处理两种不同类型的伤害：烧伤（burning）和割伤（slicing）。我们稍后将编写 Gnome.DamageType 枚举，用来代表伤害类型，几个不同的类中用来处理伤害的方法将使用这个枚举。一些类型的陷阱将应用 Burning 伤害，产生烧伤的视觉效果，其他类型的陷阱将应用 Slicing 伤害，产生相当血腥的割伤效果，从地精的身体中喷射出代表血液的红色粒子。

BodyPart 类本身被标记为需要把 SpriteRenderer 附加到游戏对象才能工作。因为不同类型的伤害会导致身体部位使用不同的精灵，所以要求有 BodyPart 脚本的任何对象也有一个 SpriteRenderer 是很合理的。

BodyPart 类存储了一些不同的属性：detachedSprite 是地精受到 Slicing 伤害时应该使用的精灵，burnedSprite 是地精受到 Burning 伤害时应该使用的精灵。另外，bloodFountainOrigin 是一个 Transform，主 Gnome 组件将用它来添加血流对象。类不是使用它，而是使用它存储的信息。

另外，BodyPart 脚本检测 RigidBody2D 组件是否已经睡眠（即已经停止移动一段时间，并且没有新的力施加到该组件）。当 RigidBody2D 组件睡眠时，BodyPart 脚本将从该组件上移除精灵渲染器之外的所有东西，这样该组件基本上就只是装饰了。必须这么处理，以免关卡中填满地精的肢体，阻碍玩家移动。



后面的 8.2 节将继续探讨血流。这里只是做一些初始的设置，让后面能够更加快速地添加血流。

接下来该添加 Gnome 脚本了。这个脚本主要是为了给后面地精的实际死亡做准备，但是提前创建好这个脚本会有帮助。

(2) **创建 Gnome 脚本**。创建一个新的 C# 脚本，命名为 Gnome.cs。

(3) 为 Gnome 组件添加代码。在 Gnome.cs 中添加下面的代码：

```
public class Gnome : MonoBehaviour {

  //摄像机应该跟随的对象
  public Transform cameraFollowTarget;

  public Rigidbody2D ropeBody;

  public Sprite armHoldingEmpty;
  public Sprite armHoldingTreasure;

  public SpriteRenderer holdingArm;

  public GameObject deathPrefab;
  public GameObject flameDeathPrefab;
  public GameObject ghostPrefab;

  public float delayBeforeRemoving = 3.0f;
  public float delayBeforeReleasingGhost = 0.25f;

  public GameObject bloodFountainPrefab;

  bool dead = false;

  bool _holdingTreasure = false;

  public bool holdingTreasure {
    get {
      return _holdingTreasure;
    }
    set {
      if (dead == true) {
        return;
      }

      _holdingTreasure = value;

      if (holdingArm != null) {
        if (_holdingTreasure) {
          holdingArm.sprite =
            armHoldingTreasure;
        } else {
          holdingArm.sprite =
            armHoldingEmpty;
        }
      }

    }
  }

  public enum DamageType {
    Slicing,
    Burning
  }
```

```
public void ShowDamageEffect(DamageType type) {
  switch (type) {

  case DamageType.Burning:
    if (flameDeathPrefab != null) {
      Instantiate(
          flameDeathPrefab,cameraFollowTarget.position,
          cameraFollowTarget.rotation
      );
    }
    break;

  case DamageType.Slicing:
    if (deathPrefab != null) {
      Instantiate(
          deathPrefab,
          cameraFollowTarget.position,
          cameraFollowTarget.rotation
      );
    }
    break;
  }
}

public void DestroyGnome(DamageType type) {

  holdingTreasure = false;

  dead = true;

  //找到全部子对象，随机断开它们的关节
  foreach (BodyPart part in
    GetComponentsInChildren<BodyPart>()) {

    switch (type) {

    case DamageType.Burning:
      //烧伤概率为1/3
      bool shouldBurn = Random.Range (0, 2) == 0;
      if (shouldBurn) {
        part.ApplyDamageSprite(type);
      }

      break;

    case DamageType.Slicing:
      //割伤总是会应用一个受伤精灵
      part.ApplyDamageSprite (type);

      break;
    }

    //从身体断开的概率为1/3
    bool shouldDetach = Random.Range (0, 2) == 0;
```

```
    if (shouldDetach) {

      //当对象停止移动以后，使其删除其刚体和碰撞器
      part.Detach ();

      //如果断开肢体，并且伤害类型为割伤，则添加血流

      if (type == DamageType.Slicing) {

        if (part.bloodFountainOrigin != null &&
          bloodFountainPrefab != null) {

          //为断开的部位添加血流
          GameObject fountain = Instantiate(
              bloodFountainPrefab,
              part.bloodFountainOrigin.position,
              part.bloodFountainOrigin.rotation
          ) as GameObject;

          fountain.transform.SetParent(
              this.cameraFollowTarget,
              false
          );
        }
      }

      //断开此对象
      var allJoints = part.GetComponentsInChildren<Joint2D>();
      foreach (Joint2D joint in allJoints) {
          Destroy (joint);
      }
    }
  }

  //向此对象添加一个RemoveAfterDelay组件
  var remove = gameObject.AddComponent<RemoveAfterDelay>();
  remove.delay = delayBeforeRemoving;

  StartCoroutine(ReleaseGhost());
}

IEnumerator ReleaseGhost() {

  //没有鬼魂预设？退出
  if (ghostPrefab == null) {
    yield break;
  }

  //等待delayBeforeReleasingGhost秒
  yield return new WaitForSeconds(delayBeforeReleasingGhost);

  //添加鬼魂
  Instantiate(
      ghostPrefab,
```

```
            transform.position,
            Quaternion.identity
        );
    }

}
```



添加这段代码时，会看到一些编译错误，包括一行或几行“无法找到名称为 RemoveAfterDelay 的类型或名称空间”。这在意料之中，我们稍后将通过添加 RemoveAfterDelay 类来解决这个问题。

Gnome 脚本主要负责保存与地精有关的重要数据，以及在地精受到伤害时进行处理。其中的许多属性并不是地精自己使用的，而是被 Game Manager（稍后将会编写）用来在需要创建新地精时进行游戏设置。

下面列出了 Gnome 脚本中需要重点注意的地方。

- 脚本中使用了一个覆写设置器来设置 holdingTreasure 属性。当 holdingTreasure 属性改变时，地精需要在视觉上发生变化：如果地精现在抱着宝藏（即 holdingTreasure 属性被设为 true），那么 Arm Holding 精灵渲染器需要变为一个包含宝藏的精灵；反之，如果属性改为 false，那么精灵渲染器需要使用一个**不**包含宝藏的精灵。
- 当地精受到伤害时，将创建一个“伤害效果”对象。具体创建什么对象，取决于受到的伤害类型。如果是 Burning，就显示一股烟。如果是 Slicing，就显示喷血效果。我们使用 ShowDamageEffect 来实现此效果。



在本书中，我们将实现喷血效果。烧伤效果留给读者完成。

- DestroyGnome 方法负责告诉所有连接的 BodyPart 组件，地精受到了伤害，它们应该断开。另外，如果伤害类型是 Slicing，那么将创建血流。
  该方法还会创建一个 RemoveAfterDelay 组件，我们稍后就添加该组件。它将整个地精从游戏中移除。
  最后，该方法启动 ReleaseGhost 协程，等待一定时间，然后创建一个 Ghost 对象。（我们将创建 Ghost 预设的工作留给读者。）

(4) **向地精的所有身体部位添加 BodyPart 脚本组件**。选择所有身体部位（头部、腿部、手臂和躯干），然后添加 BodyPart 组件。

(5) **添加血流的容器**。创建一个空游戏对象，命名为 Blood Fountains。使其成为主 Gnome 对象的子对象（即不是任何身体部位，而是父对象）。

(6) **为血流添加标记**。创建 5 个新的空游戏对象，使其成为 Blood Fountains 对象的子对象。将这些新对象按照其附加的部位命名：Head、Leg Rope、Leg Dangle、Arm Holding、Arm Loose。

移动各个对象，使它们位于你想让每个肢体的血流出现的地方（例如，将 Head 对象移动到地精的脖子位置）。然后，旋转对象，使其 *z* 轴（蓝色箭头）对准血流的喷出方向。图 5-12 给出了一个例子：选择 Head 对象，蓝色箭头指向下方。这将使血流从地精的脖子位置向上喷射。

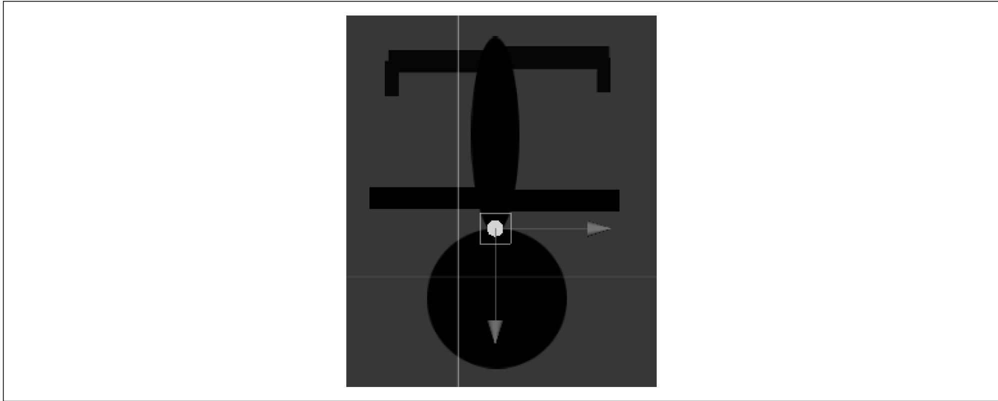


图 5-12：Head（头部）血流的位置和旋转（另见彩插）

(7) **将血流标记连接到每个身体部位**。将每个身体部位的血流拖放到 Blood Fountain Origin 框中。例如，将 Head 的 Blood Fountain Origin 游戏对象拖放到 Head 身体部位上，如图 5-13 所示。注意，Body 没有 Blood Fountain Origin 游戏对象，因为它不是可以断开的身体部位。不要将身体部位自身拖动到框中，而是推动刚才创建的新游戏对象。

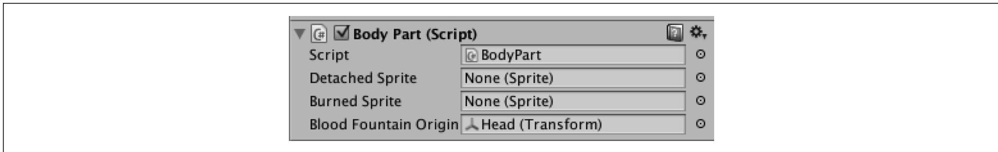


图 5-13：连接 Head（头部）的血流对象

地精的身体需要在一定延迟后消失。因此，我们将创建一个脚本，在一定时间后移除一个对象。这对于主游戏也会有帮助：就像鬼魂一样，火球也需要在一定时间后消失。

(1) **创建 `RemoveAfterDelay` 脚本**。创建一个新的 C# 脚本，命名为 RemoveAfterDelay.cs。在其中添加下面的代码：

```
//在延迟一定时间后删除对象
public class RemoveAfterDelay : MonoBehaviour {

  //在删除前等待多少秒
  public float delay = 1.0f;

  void Start () {
    //启动“删除”协程
    StartCoroutine("Remove");
  }
```

```
    IEnumerator Remove() {
      //等待delay秒，然后删除附加到此对象的gameObject
      yield return new WaitForSeconds(delay);
      Destroy (gameObject);

      //不要使用Destroy(this)，那将销毁这个RemoveAfterDelay脚本
    }
  }
```



添加了这段代码后，前面提到的编译错误将会消失：要正确编译 Gnome 类，RemoveAfterDelay 类必须存在。

RemoveAfterDelay 类很简单。当创建组件后，它使用一个协程来等待一定时间。当时间结束时，该对象将被移除。

(2) **将 Gnome 组件附加到地精**。进行如下配置。

- 将 Camera Follow Target 设置为地精的躯干。
- 将 Rope Body 设置为 Leg Rope。
- 将 Arm Holding Empty 精灵设置为 Prototype Arm Holding 精灵。
- 将 Arm Holding Treasure 精灵设置为 Prototype Arm Holding with Gold 精灵。
- 将 Holding Arm 对象设置为地精的 Arm Holding 身体部位。

完成配置后，脚本设置应如图 5-14 所示。

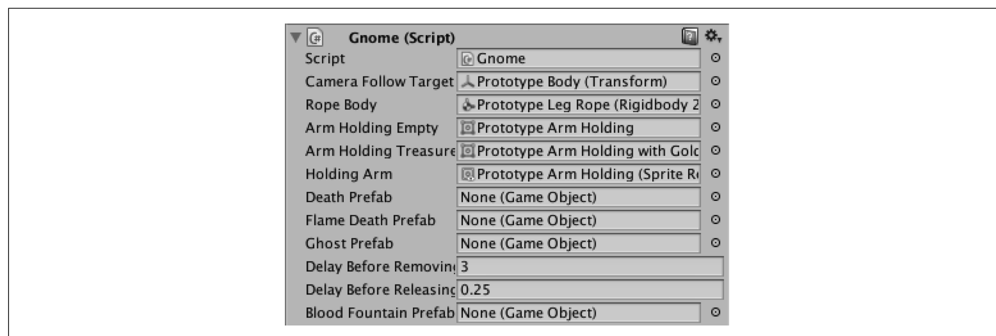


图 5-14：配置好的 Gnome 组件

我们稍后将添加 Game Manager。Game Manager 使用这些属性，让 Camera Follow 对准正确的对象，让 Rope 连接到正确的身体部位。

5.3 设置Game Manager

Game Manager 是一个对象，负责管理整个游戏。Game Manager 负责在游戏启动时创建地精，当地精触碰到重要的对象（如陷阱、宝藏或关卡出口）时进行处理，以及对生存时间

超过单个地精的所有对象进行处理。

具体来说，Game Manager 需要做以下工作。

(1) 当游戏启动或者重启时：

a. 创建地精的一个实例；
b. 如有必要，移除原有的地精；
c. 将新地精定位到关卡开始位置；
d. 将 Rope 附加到地精；
e. 使 Camera 开始跟随地精；
f. 重置所有需要重置的对象，例如宝藏。

(2) 当地精触碰到宝藏时：

通过修改地精的 `holdingTreasure` 属性，告诉它已经获得了宝藏。

(3) 当地精触碰到陷阱时：

a. 通过调用 `ShowDamageEffect`，令其显示伤害效果；
b. 通过调用 `DestroyGnome` 杀死地精；
c. 重置游戏。

(4) 当地精触碰到出口时：

如果抱着宝藏，就显示游戏结束视图。

在添加 Game Manager 的代码之前，我们需要添加 Game Manager 所依赖的一个类：`Resettable` 类。

当游戏重置时，我们想用一种普遍适用的方式来运行代码。一种方法是使用 Unity Events：我们将创建一个脚本，使其有一个 Unity Event，并将其命名为 Resettable.cs。可以把这个脚本附加到需要重置的任何对象。当游戏重置时，Game Manager 将找出所有具有 `Resettable` 组件的对象，并调用该 Unity Event。

这么做意味着我们可以配置单独的对象，使其重置自身，而不需要为每个对象编写代码。例如，我们稍后将添加的 `Treasure` 对象需要修改自己的精灵，指出已经不再有宝藏。我们将向其添加一个 `Resettable` 对象，将精灵修改为其最初的、存在宝藏的精灵。

创建 `Resettable` 脚本。添加一个新的 C# 脚本，命名为 Resettable.cs，并添加下面的代码：

```
using UnityEngine.Events;

//包含一个UnityEvent，可用于重置此对象的状态
public class Resettable : MonoBehaviour {

  //在编辑器中，将此事件连接到游戏重置时应该运行的方法
  public UnityEvent onReset;

  //游戏重置时，由GameManager调用
  public void Reset() {
    //启动事件，该事件调用所有连接的方法
```

```
        onReset.Invoke();
    }
}
```

Resettable 的代码极其简单。它只包含一个 Unity Event 属性，该属性允许在 Inspector 中添加方法调用和修改属性。当调用 Reset 方法时，将调用该事件，所添加的所有方法和属性修改将被执行。

现在就可以创建 Game Manager 了。

(1) **创建 Game Manager 对象**。创建一个新的空游戏对象，命名为 Game Manager。

(2) **创建并添加 GameManager 的代码**。为 Game Manager 对象添加一个新的 C# 脚本，命名为 GameManager.cs，在其中添加下面的代码：

```
//管理游戏状态
public class GameManager : Singleton<GameManager> {

  //地精显示的位置
  public GameObject startingPoint;

  //绳索对象，将坠下和拉起地精
  public Rope rope;

  //跟随脚本，将跟随地精
  public CameraFollow cameraFollow;

  //“当前的”地精（相对于所有死去的地精）
  Gnome currentGnome;

  //需要新地精时将实例化的预设
  public GameObject gnomePrefab;

  //UI组件，包含restart和resume按钮
  public RectTransform mainMenu;

  //UI组件，包含up、down和menu按钮
  public RectTransform gameplayMenu;

  //UI组件，包含“you win!”界面
  public RectTransform gameOverMenu;

  //如果为true，将忽略所有伤害（但仍显示伤害效果）
  //“get; set;”决定了这是一个属性，将显示在Inspector中对应于Unity事件的方法列表中
  public bool gnomeInvincible { get; set; }

  //地精死亡后等待多少秒来创建一个新地精
  public float delayAfterDeath = 1.0f;

  //地精死亡时播放的音效
  public AudioClip gnomeDiedSound;

  //玩家获胜时播放的音效
  public AudioClip gameOverSound;
```

```
void Start() {
  //游戏启动时，调用Reset来设置地精
  Reset ();
}

//重置整个游戏
public void Reset() {

  //隐藏菜单，显示游戏UI
  if (gameOverMenu)
      gameOverMenu.gameObject.SetActive(false);

  if (mainMenu)
      mainMenu.gameObject.SetActive(false);

  if (gameplayMenu)
      gameplayMenu.gameObject.SetActive(true);

  //找到所有Resettable组件，告诉它们重置
  var resetObjects = FindObjectsOfType<Resettable>();

  foreach (Resettable r in resetObjects) {
      r.Reset();
  }

  //创建一个新地精
  CreateNewGnome();

  //解除游戏暂停
  Time.timeScale = 1.0f;
}

void CreateNewGnome() {

  //如果当前有地精，就删除该地精
  RemoveGnome();

  //创建一个新的Gnome对象，使其成为currentGnome
  GameObject newGnome =
    (GameObject)Instantiate(gnomePrefab,
      startingPoint.transform.position,
      Quaternion.identity);

  currentGnome = newGnome.GetComponent<Gnome>();

  //使绳索可见
  rope.gameObject.SetActive(true);

  //将绳索的尾端连接到Gnome对象
  //想要连接的刚体（如地精的脚）
  rope.connectedObject = currentGnome.ropeBody;

  //将绳索的长度重置为默认值
  rope.ResetLength();
```

```
        //告诉cameraFollow开始跟踪新的Gnome对象
        cameraFollow.target = currentGnome.cameraFollowTarget;

    }

    void RemoveGnome() {

        //如果地精处于无敌状态，那么什么都不做
        if (gnomeInvincible)
            return;

        //隐藏绳索
        rope.gameObject.SetActive(false);

        //停止跟随地精
        cameraFollow.target = null;

        //如果有一个当前的地精，使该地精不再代表玩家
        if (currentGnome != null) {

            //此地精不再抱着宝藏
            currentGnome.holdingTreasure = false;

            //将此对象标记为不是玩家（这样此对象与碰撞器发生碰撞时，
            //碰撞器不会给出报告）
            currentGnome.gameObject.tag = "Untagged";

            //找到当前所有带有Player标记的对象，并删除该标记
            foreach (Transform child in
                currentGnome.transform) {
                    child.gameObject.tag = "Untagged";
            }

            //把我们标记为当前没有地精
            currentGnome = null;
        }
    }

    //杀死地精
    void KillGnome(Gnome.DamageType damageType) {

            //如果有音效源，就播放“地精死亡”音效
            var audio = GetComponent<AudioSource>();
            if (audio) {
                    audio.PlayOneShot(this.gnomeDiedSound);
            }

            //显示伤害效果
            currentGnome.ShowDamageEffect(damageType);

            //如果不是无敌的，就重置游戏，使该地精不再代表当前玩家
            if (gnomeInvincible == false) {

                    //告诉地精已死亡
                    currentGnome.DestroyGnome(damageType);
```

```
            //删除地精
            RemoveGnome();

            //重置游戏
            StartCoroutine(ResetAfterDelay());

        }
    }

    //地精死亡时调用
    IEnumerator ResetAfterDelay() {

        //等待delayAfterDeath秒，然后调用Reset
        yield return new WaitForSeconds(delayAfterDeath);
        Reset();
    }

    //当玩家触碰到陷阱时调用
    public void TrapTouched() {
      KillGnome(Gnome.DamageType.Slicing);
    }

    //当玩家触碰到喷火陷阱时调用
    public void FireTrapTouched() {
      KillGnome(Gnome.DamageType.Burning);
    }

    //当玩家捡起宝藏时调用
    public void TreasureCollected() {
        //告诉currentGnome已经捡起宝藏
        currentGnome.holdingTreasure = true;
    }

    //当玩家触碰到出口时调用
    public void ExitReached() {
      //如果有玩家，并且该玩家抱着宝藏，则游戏结束
      if (currentGnome != null &&
        currentGnome.holdingTreasure == true) {

        //如果有音效源，则播放Game Over音效
        var audio = GetComponent<AudioSource>();
        if (audio) {
          audio.PlayOneShot(this.gameOverSound);
        }

        //暂停游戏
        Time.timeScale = 0.0f;

        //关闭Game Over菜单，显示Game Over画面
        if (gameOverMenu) {
          gameOverMenu.gameObject.SetActive(true);
        }

        if (gameplayMenu) {
```

```
            gameplayMenu.gameObject.SetActive(false);
          }

        }
      }

      //当触摸Menu菜单时，以及触摸Resume Game按钮时调用
      public void SetPaused(bool paused) {

        //如果已经暂停，就停止时间并启用菜单（以及禁用游戏叠加画面）
        if (paused) {
          Time.timeScale = 0.0f;
          mainMenu.gameObject.SetActive(true);
          gameplayMenu.gameObject.SetActive(false);
        } else {
          //如果没有暂停，就恢复时间并启用菜单（以及启用游戏叠加画面）
          Time.timeScale = 1.0f;
          mainMenu.gameObject.SetActive(false);
          gameplayMenu.gameObject.SetActive(true);
        }
      }

      //当触摸Restart按钮时调用
      public void RestartGame() {

        //立即移除地精（而不是杀死地精）
        Destroy(currentGnome.gameObject);
        currentGnome = null;

        //重置游戏以创建新地精
        Reset();
      }

    }
```

Game Manager 主要为创建新地精而设计，以及用于将其他系统连接到正确的对象。当需要生成新的地精时，需要把 Rope 对象连接到新地精的腿部，并将 `CameraFollow` 指向地精的躯干。Game Manager 还负责处理菜单的显示，以及响应这些菜单的按钮（我们将在以后实现菜单）。

因为这段代码很多，我们将详细解释其作用。

5.3.1 设置和重置游戏

对象第一次出现时将调用 `Start` 方法，它会立即调用 `Reset` 方法。`Reset` 的作用是将整个游戏重置到初始状态，在 `Start` 中调用 `Reset` 能够将"初始设置"和"重置游戏"的代码快速合并在一个位置。

`Reset` 方法确保合适的菜单元素（我们将在后面设置）是可见的。`Reset` 方法将告诉场景中的全部 `Resettable` 组件重置，并调用 `CreateNewGnome` 方法来创建一个新的地精。最后，`Reset` 方法解除游戏的暂停状态（只是防止游戏**之前**被暂停了）。

```
void Start() {
  //游戏启动时，调用Reset来设置地精
  Reset ();
}

//重置整个游戏
public void Reset() {

  //隐藏菜单，显示游戏UI
  if (gameOverMenu)
      gameOverMenu.gameObject.SetActive(false);

  if (mainMenu)
      mainMenu.gameObject.SetActive(false);

  if (gameplayMenu)
      gameplayMenu.gameObject.SetActive(true);

  //找到所有Resettable组件，告诉它们重置
  var resetObjects = FindObjectsOfType<Resettable>();

  foreach (Resettable r in resetObjects) {
      r.Reset();
  }

  //创建一个新地精
  CreateNewGnome();

  //解除游戏暂停
  Time.timeScale = 1.0f;
}
```

5.3.2 创建新地精

`CreateNewGnome` 方法用一个新构造的地精替换了原来的地精。该方法首先移除当前的地精（如果存在的话），然后创建一个新地精。它还启用绳索，并将地精的脚踝（它的 `ropeBody`）连接到绳索的末端。然后，告诉绳索将其长度重置为初始值。最后，使摄像机跟随新地精：

```
void CreateNewGnome() {

    //如果当前有地精，就删除该地精
    RemoveGnome();

    //创建一个新的Gnome对象，使其成为currentGnome
    GameObject newGnome =
      (GameObject)Instantiate(gnomePrefab,
        startingPoint.transform.position,
        Quaternion.identity);

    currentGnome = newGnome.GetComponent<Gnome>();
```

```
    //使绳索可见
    rope.gameObject.SetActive(true);

    //将绳索的尾端连接到Gnome对象想要连接的刚体（如地精的脚）
    rope.connectedObject = currentGnome.ropeBody;

    //将绳索的长度重置为默认值
    rope.ResetLength();

    //告诉cameraFollow开始跟踪新的Gnome对象
    cameraFollow.target = currentGnome.cameraFollowTarget;

}
```

5.3.3 移除旧地精

在两种情况下，我们需要把地精与绳索断开：地精死亡时，或者玩家决定开始新一局游戏时。在这两种情况下，旧地精断开连接，也不再代表玩家。它仍然在关卡内，但是即使触碰到陷阱，游戏也不会将其解读为重新开始关卡的信号。

为了移除活动的地精，我们禁用绳索，并使摄像机停止跟随当前的地精。然后，我们将此地精标记为不再抱着宝藏，这会使其精灵切换回常规版本，并将该对象标记为 Untagged。之所以这么做，是因为我们稍后添加的陷阱会寻找标记为 Player 的对象。如果旧地精仍被标记为 Player，那么陷阱将通知 Game Manager 重新开始关卡。

```
void RemoveGnome() {

  //如果地精处于无敌状态，那么什么都不做
  if (gnomeInvincible)
    return;

  //隐藏绳索
  rope.gameObject.SetActive(false);

  //停止跟随地精
  cameraFollow.target = null;

  //如果有一个当前的地精，使该地精不再代表玩家
  if (currentGnome != null) {

    //该地精不再抱着宝藏
    currentGnome.holdingTreasure = false;

    //将该对象标记为不是玩家（这样，当其与碰撞器发生碰撞时，
    //碰撞器不会给出报告）
    currentGnome.gameObject.tag = "Untagged";

    //找到当前所有带有Player标记的对象，并删除该标记
    foreach (Transform child in
      currentGnome.transform) {
        child.gameObject.tag = "Untagged";
    }
```

```
        //把我们标记为当前没有地精
        currentGnome = null;
    }
}
```

杀死地精

当地精被杀死后，我们需要在游戏中显示合适的效果，包括音效和特殊效果。另外，如果地精当前不是无敌状态，我们应该告诉地精其已经死亡，移除地精，延迟一定时间后重置游戏。下面是实现上述效果的代码：

```
void KillGnome(Gnome.DamageType damageType) {

    //如果有音效源，就播放“地精死亡”音效
    var audio = GetComponent<AudioSource>();

    if (audio) {
        audio.PlayOneShot(this.gnomeDiedSound);
    }

    //显示伤害效果
    currentGnome.ShowDamageEffect(damageType);

    //如果地精不是无敌的，就重置游戏，使该地精不再代表当前的玩家
    if (gnomeInvincible == false) {

        //告诉地精已死亡
        currentGnome.DestroyGnome(damageType);

        //删除地精
        RemoveGnome();

        //重置游戏
        StartCoroutine(ResetAfterDelay());

    }
}
```

5.3.4 重置游戏

当地精死亡后，我们想让摄像机在地精死亡的位置停留一会儿。这样一来，在返回屏幕顶部之前，玩家能够看到地精向下坠落。

为此，我们使用一个协程来等待几秒（存储在 `delayAfterDeath` 中），然后调用 `Reset` 来重置游戏状态：

```
//地精死亡时调用
IEnumerator ResetAfterDelay() {

    //等待delayAfterDeath秒，然后调用Reset
    yield return new WaitForSeconds(delayAfterDeath);
    Reset();

}
```

5.3.5 处理触碰

接下来的 3 个方法对地精触碰特定对象的事件进行处理。如果地精触碰到一个陷阱，那就调用 `killGnome`，指出发生了割伤。如果地精触碰到一个喷火陷阱，就指出发生了烧伤。最后，如果收集到宝藏，就让地精开始抱着该宝藏。下面是实现上述效果的代码：

```
//当玩家触碰到陷阱时调用
public void TrapTouched() {
  KillGnome(Gnome.DamageType.Slicing);
}

//当玩家触碰到喷火陷阱时调用
public void FireTrapTouched() {
  KillGnome(Gnome.DamageType.Burning);
}

//当地精捡起宝藏时调用
public void TreasureCollected() {
  //告诉currentGnome已经捡起宝藏
  currentGnome.holdingTreasure = true;
}
```

5.3.6 到达出口

当地精触碰到关卡顶端的出口时，我们需要检查当前的地精是否抱着宝藏。如果抱着，那么玩家获胜！我们将播放一个 Game Over 音效（8.4 节将设置），通过将 `timeScale` 设为 `0` 来暂停游戏，并显示 Game Over 画面（包含一个重置游戏的按钮）：

```
//当玩家触碰到出口时调用
public void ExitReached() {
  //如果有玩家，并且该玩家抱着宝藏，则游戏结束
  if (currentGnome != null &&
    currentGnome.holdingTreasure == true) {

    //如果有音效源，则播放Game Over音效
    var audio = GetComponent<AudioSource>();
    if (audio) {
      audio.PlayOneShot(this.gameOverSound);
    }

    //暂停游戏
    Time.timeScale = 0.0f;

    //关闭Game Over菜单，显示Game Over画面
    if (gameOverMenu) {
      gameOverMenu.gameObject.SetActive(true);
    }

    if (gameplayMenu) {
      gameplayMenu.gameObject.SetActive(false);
    }
  }
}
```

5.3.7 暂停与恢复

暂停游戏涉及以下 3 个操作。首先，通过把 `timeScale` 设为 `0` 来停止时间。然后，使主菜单可见，并隐藏游戏 UI。要恢复运行游戏，只需执行反向操作：将时间设为再次前进，隐藏主菜单，并显示游戏 UI。

```
//当触摸Menu菜单时，以及触摸Resume Game按钮时调用
public void SetPaused(bool paused) {

  //如果已经暂停，就停止时间并启用菜单（以及禁用游戏叠加画面）
  if (paused) {
    Time.timeScale = 0.0f;
    mainMenu.gameObject.SetActive(true);
    gameplayMenu.gameObject.SetActive(false);
  } else {
    //如果没有暂停，就恢复时间并启用菜单（以及启用游戏叠加画面）
    Time.timeScale = 1.0f;
    mainMenu.gameObject.SetActive(false);
    gameplayMenu.gameObject.SetActive(true);
  }
}
```

5.3.8 处理Reset按钮

当用户在 UI 中单击特定的按钮时，将调用 `RestartGame` 方法。该方法将立即重新开始游戏：

```
//当触摸Restart按钮时调用
public void RestartGame() {

  //立即移除地精（而不是杀死地精）
  Destroy(currentGnome.gameObject);
  currentGnome = null;

  //重置游戏以创建新地精
  Reset();
}
```

5.4 准备场景

编写好代码后，我们可以按如下步骤设置场景来使用代码。

(1) **创建起点**。这是一个对象，Game Manager 使用它来放置新创建的地精。创建一个新的游戏对象，命名为 Start Point。将其放到你想让地精一开始出现的地方（靠近 Rope 的地方就可以，如图 5-15 所示），并将其图标修改为胶囊形（就像设置 Rope 的图标那样）。

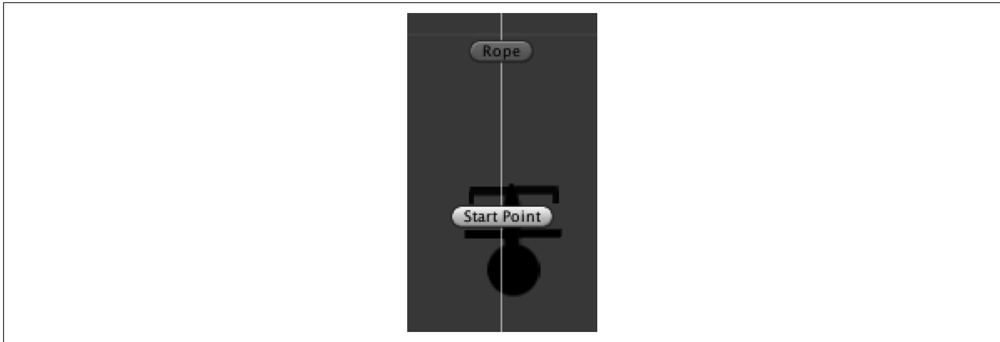


图 5-15：确定起点的位置

(2) **将地精转换为一个预设**。现在地精将由 Game Manager 创建，这意味着需要移除当前位于场景中的地精。在移除之前，需要将其转换为一个预设，这样 Game Manager 就可以在运行时创建它的实例。

将地精拖放到 Project 窗格的 Gnome 文件夹中。这将创建一个新的预设对象（如图 5-16 所示），它完全是原 Gnome 对象的复制品。

现在已经创建了预设，就不再需要场景中的对象了。将地精从场景中移除。

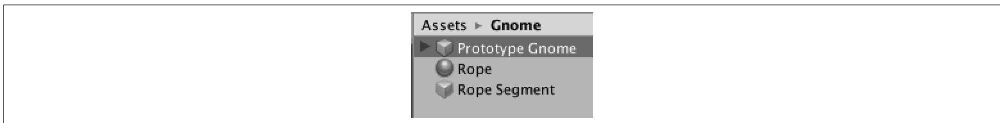


图 5-16：地精成为了 Gnome 文件夹中的一个预设

(3) **配置 Game Manager**。我们需要为 Game Manager 设置几个如下的连接。

- 将 Starting Point 字段连接到刚才创建的 Start Point 对象。
- 将 Rope 字段连接到 Rope 对象。
- 将 Camera Follow 字段连接到 Main Camera。
- 将 Gnome Prefab 字段连接到刚才创建的 Gnome 预设。

完成上述操作后，Game Manager 的 Inspector 将如图 5-17 所示。

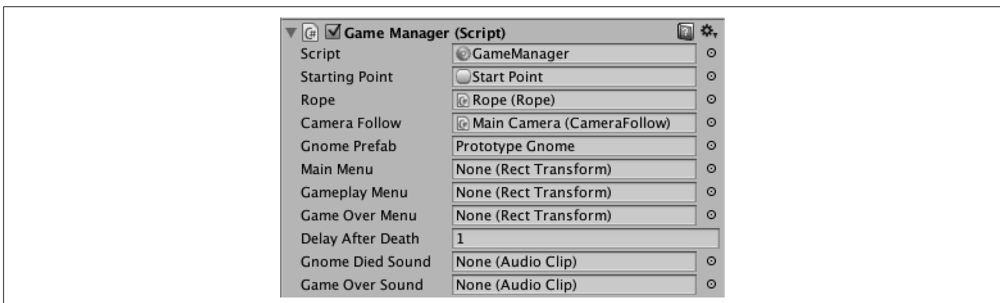


图 5-17：Game Manager 的设置

(4) **测试游戏**。地精将出现在起点，并连接到绳索。另外，当拉起或放下绳索时，摄像机将跟随地精的身体移动。现在还无法测试抱着宝藏的情况，但是不要担心，我们很快就会处理。

5.5 小结

创建好 Game Manager 以后，就可以添加游戏的实际玩法了。第 6 章里我们将开始添加与地精交互的元素：宝藏和陷阱。

第6章

使用陷阱和目标建立游戏玩法

我们已经建立了游戏玩法的基础，现在可以开始加入游戏元素，例如陷阱和宝藏。之后，游戏的其余部分主要就是关卡设计了。

6.1 简单的陷阱

这个游戏主要处理的行为是玩家触碰到对象——陷阱、宝藏、出口等——的时间。因为检测玩家何时触碰到特定的对象非常重要，所以我们将创建一个通用的脚本，当地精碰撞到任何标记为Player的对象时，将触发一个Unity Event。针对不同的对象，可以采用不同的方式设置此事件：陷阱可以配置为，告诉Game Manager，地精已经受到了伤害；宝藏可以配置为，告诉Game Manager，地精已经获得了宝藏；出口可以配置为，告诉Game Manager，地精已经到达了出口。

现在，创建一个新的C#脚本，命名为SignalOnTouch.cs，在其中添加下面的代码：

```
using UnityEngine.Events;

//当玩家与此对象发生碰撞时调用UnityEvent
[RequireComponent (typeof(Collider2D))]
public class SignalOnTouch : MonoBehaviour {

  //发生碰撞时调用的UnityEvent
  //在编辑器中附加要运行的方法
  public UnityEvent onTouch;

  //如果为true，则在发生碰撞时试图播放AudioSource
  public bool playAudioOnTouch = true;

  //进入触发器区域时，调用SendSignal
  void OnTriggerEnter2D(Collider2D collider) {
    SendSignal (collider.gameObject);
```

```
    }

    //与此对象发生碰撞时，调用SendSignal
    void OnCollisionEnter2D(Collision2D collision) {
      SendSignal (collision.gameObject);
    }

    //检查此对象是否被标记为Player，如果是，就调用UnityEvent
    void SendSignal(GameObject objectThatHit) {

      //此对象被标记为Player了吗？
      if (objectThatHit.CompareTag("Player")) {

        //如果应该播放音效，则尝试播放
        if (playAudioOnTouch) {
          var audio = GetComponent<AudioSource>();

          //如果有一个音效组件，并且此组件的父组件是活跃的，
          //那么就播放此音效
          if (audio &&
            audio.gameObject.activeInHierarchy)
            audio.Play();
        }

        //调用事件
        onTouch.Invoke();
      }
    }

  }
```

SignalOnTouch 类的主要代码在 SendSignal 方法中处理，后者由 OnCollisonEnter2D 和 OnTriggerEnter2D 调用。当对象触碰到碰撞器，或者当对象进入触发器时，Unity 将分别调用前述的两个方法。SendSignal 方法检查碰撞对象的标记，如果是 Player，就调用 Unity Event。

现在已经准备好了 SignalOnTouch 类，可以添加第一个陷阱了。

(1) **导入关卡对象精灵**。将 Sprites/Objects 文件夹的内容导入到项目中。
(2) **添加棕色尖刺**。找到 SpikesBrown 精灵，将其拖放到场景中。
(3) **配置尖刺对象**。向尖刺添加一个 Polygon Collider 2D 组件，以及一个 Signal On Touch 组件。向 Signal On Touch 的事件添加一个新函数。将 Game Manager 拖放到对象框中，并将函数设置为 GameManager.TrapTouched，如图 6-1 所示。



图 6-1：设置尖刺

(4) **将尖刺转换为预设**。将 SpikesBrown 对象从 Hierarchy 拖放到 Level 文件夹。这将创建一个预设，意味着可以生成对象的多个副本。
(5) **进行测试**。运行游戏，使地精触碰到尖刺。地精将掉出摄像机镜头以外，然后重新生成。

6.2 宝藏和出口

现在已经成功添加了一种杀死地精的方式，是时候添加一种让玩家获胜的方式了。这通过添加两个新的对象——宝藏和出口——来实现。

宝藏是位于井底的一个精灵，它会检测玩家何时触碰到自己，并通知 Game Manager。当收到信号时，Game Manager 将通知地精它已经获得宝藏，于是地精的手臂精灵会改为抱着宝藏的样子。

出口是另外一个精灵，位于井口。与宝藏一样，它会检测玩家何时触碰到自己，并通知 Game Manager。如果地精在触碰到出口时携带宝藏，那么玩家获胜。

关于这两个对象的大部分工作是由 Signal On Touch 组件处理的——当触碰到出口时，需要调用 Game Manager 的 `ExitReached` 方法；当触碰到宝藏时，需要调用 Game Manager 的 `TreasureCollected` 方法。

我们首先创建出口，然后添加宝藏。

创建出口

先来按如下步骤导入精灵。

(1) **导入 Level Background 精灵**。将 Sprites/Background 文件夹从下载的资源中导入 Sprites 文件夹。

(2) **添加 Top 精灵**。将其放到稍微低于 Rope 对象的地方，它将作为 Exit（出口）。

(3) **配置精灵**。向精灵添加一个 Box Collider 2D 组件，打开其 Is Trigger 属性。单击 Edit Collider 按钮，调整框的大小，使其短而宽（如图 6-2 所示）。

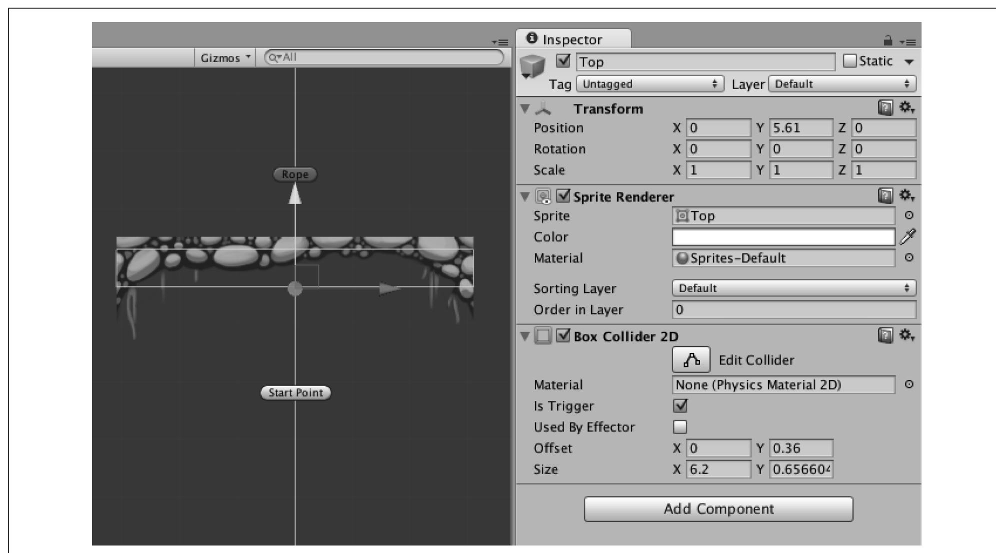


图 6-2：将出口的碰撞器设置为短而宽的形状，并放到关卡上方

(4) **让精灵被触碰到时通知 Game Manager**。向精灵添加一个 Signal On Touch 组件。在组件的事件中添加一个条目，并将其连接到 Game Manager。将函数设为 `GameManager.ExitReached`。这样，当地精触碰到精灵时，Game Manager 的 `ExitReached` 方法将会运行。

接下来要添加宝藏。

宝藏的工作方式如下。默认情况下，Treasure 对象显示一个宝藏精灵。当玩家触碰到宝藏时，将调用 Game Manager 的 `TreasureCollected` 方法，宝藏的精灵将显示宝藏已被获得。如果地精死亡，那么 Treasure 对象将重置为包含宝藏的精灵。

因为在游戏接下来的部分，特别是到了优化阶段，将一个精灵换为另一个精灵是很寻常的操作，所以应该创建一个通用的精灵切换类，并将宝藏设置为使用这个切换类。

创建一个新的 C# 脚本，命名为 SpriteSwapper.cs，在其中添加下面的代码：

```
//将精灵切换为另外一个精灵。例如，将宝藏精灵
//从treasure present换为treasure not present
public class SpriteSwapper : MonoBehaviour {

  //应该显示的精灵
  public Sprite spriteToUse;

  //应该使用新精灵的精灵渲染器
  public SpriteRenderer spriteRenderer;

  //原来的精灵，调用ResetSprite时使用
  private Sprite originalSprite;

  //换出精灵
  public void SwapSprite() {

    //如果此精灵不同于当前的精灵……
    if (spriteToUse != spriteRenderer.sprite) {

      //将之前的精灵保存到originalSprite中
      originalSprite = spriteRenderer.sprite;

      //使精灵渲染器使用新精灵
      spriteRenderer.sprite = spriteToUse;
    }
  }

  //恢复使用原来的精灵
  public void ResetSprite() {

    //如果之前有一个精灵……
    if (originalSprite != null) {
      //……则让精灵渲染器使用它
      spriteRenderer.sprite = originalSprite;
    }
  }
}
```

`SpriteSwapper` 类会完成两个操作。当调用 `SwapSprite` 方法时，将告知附加到游戏对象的 `SpriteRenderer` 改变其精灵；另外，原来的精灵将被存储到一个变量中。当调用 `ResetSprite` 方法时，精灵渲染器将被恢复为使用原来的精灵。

现在可以创建并设置 Treasure 对象。

(1) **添加宝藏精灵**。找到 TreasurePresent 精灵，将其添加到场景中。将其放到井底的某个位置，但要确保地精仍然能够触碰到宝藏。

(2) **为宝藏添加一个碰撞器**。选择宝藏精灵，添加一个 Box Collider 2D，使此碰撞器成为一个触发器。

(3) **添加并配置一个精灵切换器**。添加一个 Sprite Swapper 组件。将宝藏精灵自身拖放到 Sprite Renderer 字段中。接下来，找到 TreasureAbsent 精灵，将其拖放到精灵切换器的 Sprite To Use 字段中。

(4) **添加并配置一个"触碰时发出信号"组件**。添加一个 Signal On Touch 组件。在 On Touch 列表中添加如下两个条目。

- 首先，连接 Game Manager 对象，并使事件的方法是 `GameManager.TreasureCollected`。
- 接下来，连接宝藏精灵（即当前配置的对象），并使其方法是 `SpriteSwapper.SwapSprite`。

(5) **添加并配置一个 Resettable 组件**。向对象添加一个 Resettable 组件。在 On Touch 方法中添加一个条目，设方法为 `SpriteSwapper.ResetSprite`，并将其连接到 Treasure 对象。

完成上述操作后，Treasure 对象的 Inspector 应如图 6-3 所示。

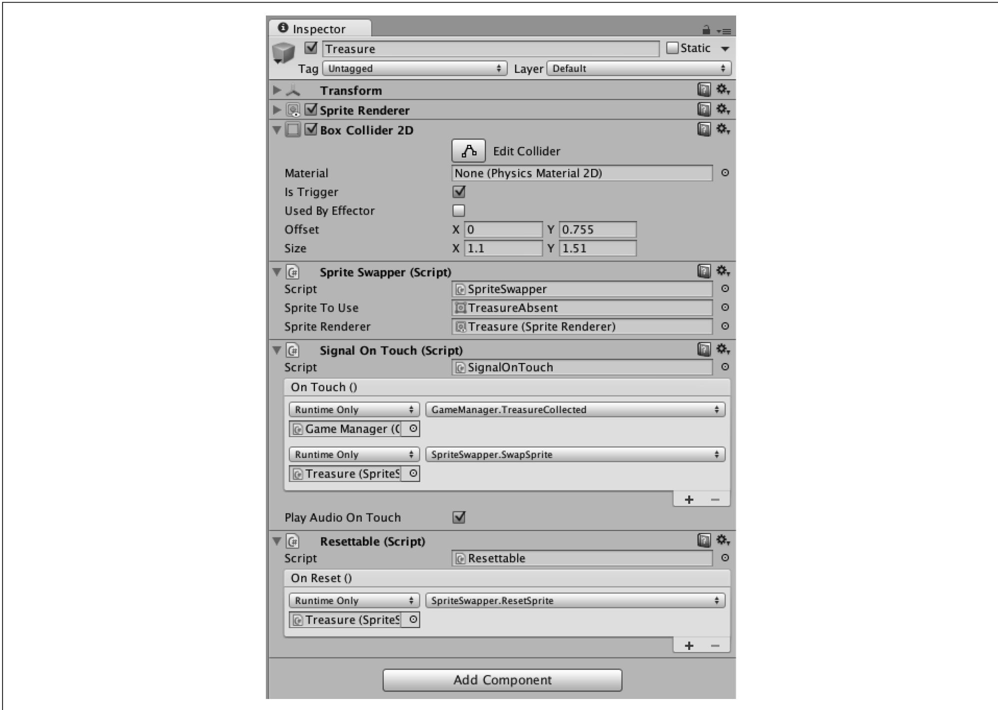


图 6-3：配置好的 Treasure 对象

(6) **测试游戏**。运行游戏，并触碰宝藏。当触碰到宝藏时，宝藏会消失。如果玩家死亡，那么当地精重新出现时，宝藏也将重新出现。

6.3　添加背景

目前地精处于默认的 Unity 背景之上，这是一种不怎么好看的蓝色。我们将添加一个临时的背景，当后面开始优化画面时，再用一个背景精灵替换这个临时的背景。

(1) **添加背景四边形**。打开 GameObject 菜单，选择 3D Object → Quad。将新对象命名为 Background。

(2) **将背景移动得更远一些**。为了避免背景四边形出现在游戏精灵的前方，让背景四边形远离摄像机。将背景四边形位置的 *z* 值设为 10。



虽然这是一个 2D 游戏，但是 Unity 仍然是一个 3D 引擎。这意味着某些对象位于其他对象“后面”的概念仍然是存在的，此处就利用了这个事实。

(3) **确定背景四边形的位置**。按 T 键切换到 Rect 工具，然后使用尺寸调整手柄来调整背景四边形的大小，使背景的顶边与关卡顶部的精灵对齐，底边与宝藏对齐，如图 6-4 所示。



图 6-4：调整背景四边形的大小

(4) **测试游戏**。玩游戏时，关卡将有一个灰色的背景。

6.4 小结

到了构建游戏的这一步，核心的游戏玩法功能已经具备了。我们添加了很多玩法，列举如下。

- 地精模拟了物理效果，并将其附加到一条同样模拟了物理效果的绳索上。
- 通过屏幕上的按钮可控制绳索，这意味着可以放下和拉起地精。
- 设置了摄像机来跟随地精的位置，使其在整个游戏过程中始终可见。
- 地精响应手机的倾斜动作，并能左右晃动。
- 地精碰到陷阱时会被杀死。地精能够收集宝藏。

图 6-5 是游戏在当前状态下的一个屏幕截图。



图 6-5：本章结束时的游戏

虽然游戏的功能已经完善，但现在看起来并不是很漂亮。地精仍然只是一个火柴人，关卡也很简略。第 7 章中，我们将继续构建游戏，并改善屏幕上每个元素的视觉效果。

第 7 章

优化游戏

到本章结束时，我们将对 *Gnome's Well That Ends Well* 做出大量调整，最终结果将如图 7-1 所示。

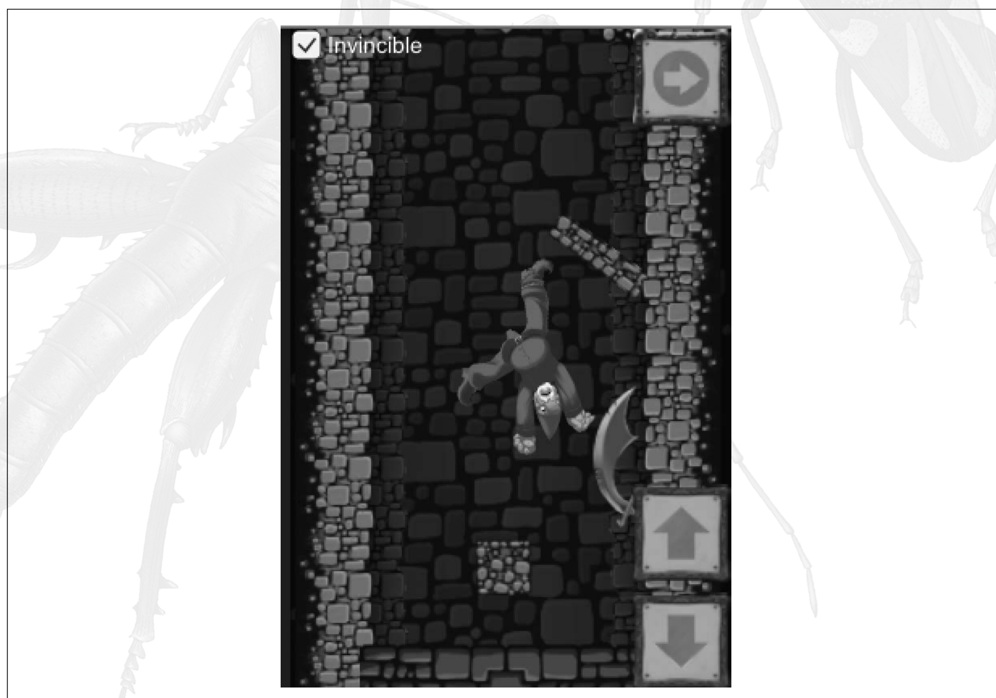


图 7-1：最终的游戏效果

我们将对游戏的 3 个主要方面做出优化。

画面优化
: 我们将为地精添加新的精灵，改善背景的外观，并添加粒子效果来改善游戏的画面。

游戏玩法优化
: 我们将添加不同类型的陷阱，一个标题画面，以及一种让地精变成无敌状态的方法，这有助于测试游戏玩法。

音效优化
: 我们还将为游戏添加音效，响应玩家的操作。

本章用到的资源可在该地址的资源包中找到：https://www.secretlab.com.au/books/unity。

7.1 更新游戏的画面

优化游戏首先要做的是修改地精的精灵，把它们从目前的火柴人替换为一组手绘的精灵。

为此，将 GnomeParts 文件夹从原始资源复制到 Sprites 文件夹。此文件夹包含两个子文件夹：Alive（生存）文件夹包含地精的新部位，Dead（死亡）文件夹包含当地精死亡时使用的精灵（如图 7-2 所示）。我们先来处理生存精灵，以后会用到死亡精灵。

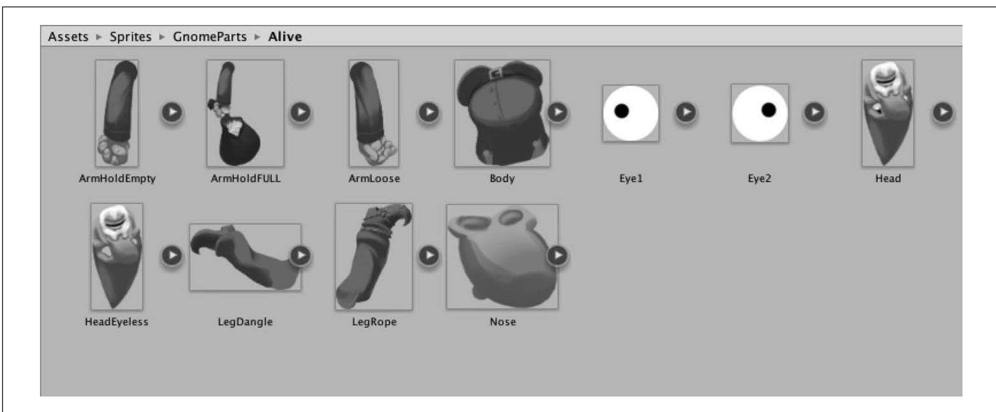


图 7-2：地精的生存精灵（另见彩插）



我们不会用到下载的全部资源，例如资源中还有一个没有眼睛的头部精灵，它要与另外的眼睛精灵结合使用。如果想要在本书的基础上进一步拓展游戏，你可能会用到这些额外的资源。

下面的第一步是配置精灵，使它们能够用在 Gnome 对象中。尤其要注意的是，我们要确保把它们作为精灵导入，并且精灵的锚点处于正确的位置。需要执行的步骤如下。

(1) **如果图片还不是精灵，则转换为精灵**。选择 Alive 文件夹中的精灵，确保将纹理类型设为 Sprite (2D and UI)。

(2) **更新精灵的锚点**。对于除 Body 之外的每个精灵，执行以下操作。

a. 选择精灵。
b. 单击 Sprite Editor 按钮。
c. 拖放锚点图标（小蓝圈）到合适的位置，身体部位将围绕这个点旋转。例如，图 7-3 显示了 ArmHoldEmpty 精灵的位置。

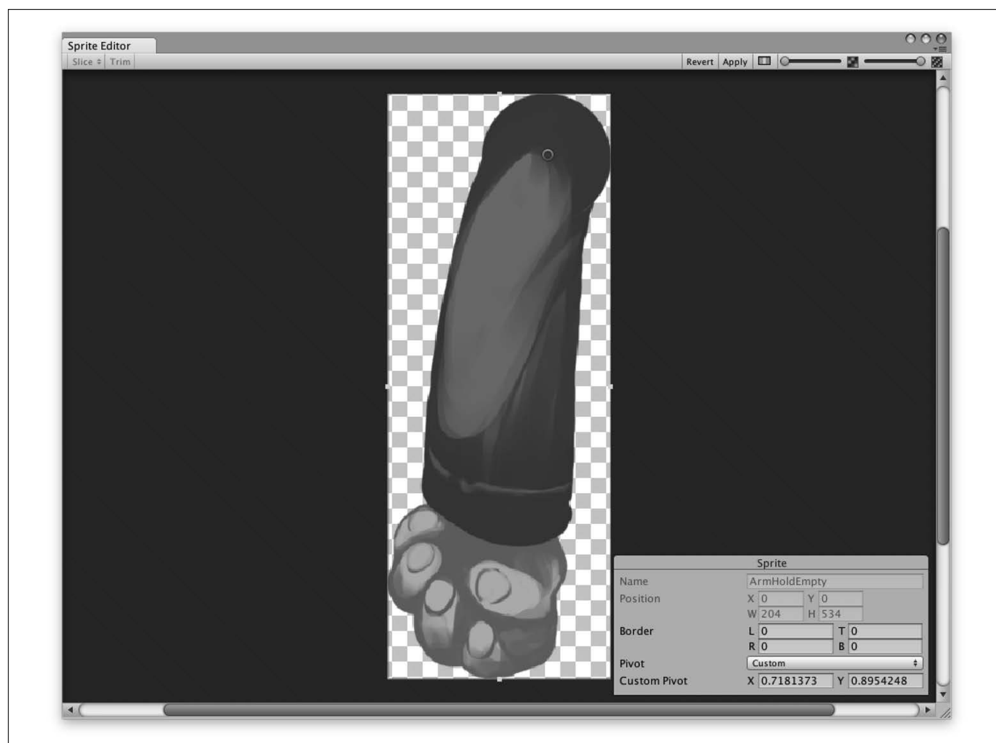


图 7-3：设置 ArmHoldEmpty 精灵的锚点；注意锚点的位置位于右上角

设置好精灵后，就要把它们添加到地精中。为了保持整齐有序，并保留旧版本的地精，我们将复制地精预设，并在新版本上进行修改。然后，我们告诉游戏使用这个新的、改进过的版本。

完成上述操作后，我们将把新地精添加到场景中，并使用新的美术作品替换地精身体的各个组件。美极了，妙极了。具体操作如下。

(1) **复制原型地精预设**。找到原型地精预设，按 Ctrl-D 键进行复制（在 Mac 上是 Command-D 键）。将新对象命名为 Gnome。
(2) **把新地精添加到场景中**。将新的地精预设拖动到场景窗口中，以创建一个新的实例。
(3) **替换美术作品**。选择地精的全部身体部位，替换为对应的新精灵。例如，选择头部，将其精灵替换为 Alive 文件夹中的 Head 精灵。

完成之后，地精应该如图 7-4 所示。身体部位的位置不够精确，不过这不是问题，后面会进行调整。

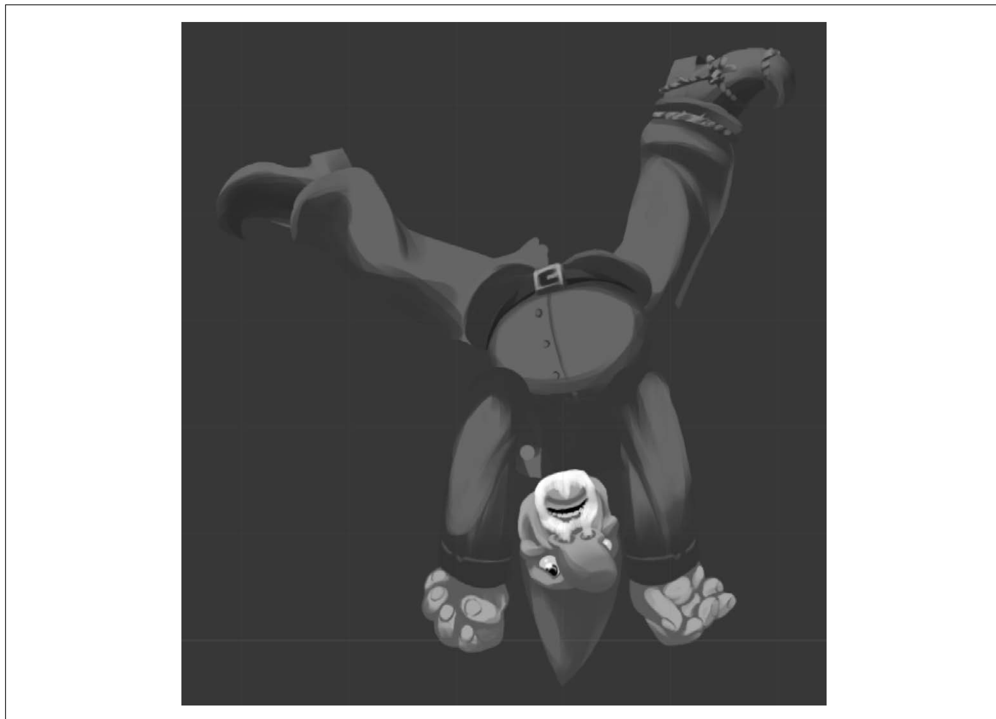


图 7-4：更新精灵后的 Gnome 对象

接下来，我们需要调整地精身体部位的位置。新精灵的形状和大小各异，我们需要让各个部位准确排列。执行下面的设置步骤。

(1) **调整头部、手臂和腿部的位置**。选择 Head 对象，让脖子位于肩膀中间的正确位置。在手臂（与肩膀对齐）和腿部（与腰部对齐）之上重复这个过程。

注意，肩膀的轴点在 Body 精灵上的紫色点位置。

调整位置后，需要确保精灵总是遵守正确的顺序：腿部不应该绘制到躯干的前方，躯干不应该绘制到手臂的前方，头部应该在其他身体部位的前方。

(2) **调整身体部位的排序**。选择头部和双臂，将 Sprite Renderer 的 Order in Layer 属性改为 2。接下来，选择躯干，将其顺序改为 1。

完成之后，精灵应该如图 7-5 所示。



图 7-5：设置了正确的精灵位置之后的地精

7.2　更新物理组件

更新了地精的精灵后，还要更新物理组件。我们要修改两个地方：更新碰撞器，使其匹配正确的形状；调整关节，使身体部位的轴点处于正确的位置。

首先调整碰撞器。因为精灵不是水平或者垂直的线条，我们需要把当前的矩形和圆形碰撞器替换为**多边形碰撞器**。

创建多边形碰撞器的方法有两种：让 Unity 替你生成一个形状，或者自己指定形状。我们选择自己指定，因为这种方法更加高效（Unity 一般会生成复杂的形状，对于性能而言不太有利），而且可以让我们更好地控制最终结果。

为具有精灵渲染器的对象添加多边形碰撞器时，Unity 将使用精灵，通过绘制一条围绕图片所有不透明部分的线来构建多边形。如果想要定义自己的碰撞形状，需要把多边形碰撞器组件添加到一个**没有**精灵渲染器的游戏对象上。最简单的方式是创建一个空的子对象，为其添加一个多边形碰撞器。为此，需要执行下面的步骤。

(1) **移除已有的碰撞器**。选择全部的腿部和手臂，移除 Box Collider 2D。接下来，选择头部，移除 Circle Collider 2D。

(2) 为每个手臂、每个腿部和头部重复下面的步骤。

  a. **为碰撞器添加子对象**。创建新的空游戏对象，命名为 Collider。使其成为身体部位的子对象，并确保其位置为 (0,0,0)。

  b. **添加多边形碰撞器**。选择此新的 Collider 对象，为其添加一个 Polygon Collider 2D 组件。此时将出现一个碰撞器的形状，如图 7-6 所示。默认情况下，Unity 将为其创建一个多边形，但是需要调整它来适应对象。

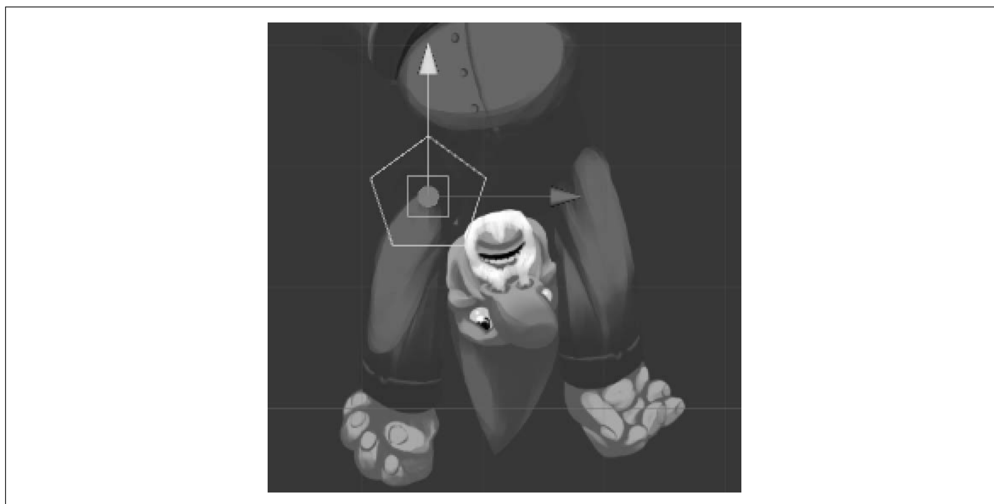


图 7-6：新添加的 Polygon Collider 2D

  c. **编辑多边形碰撞器的形状**。单击 Edit Collider 按钮（如图 7-7 所示），进入编辑模式。

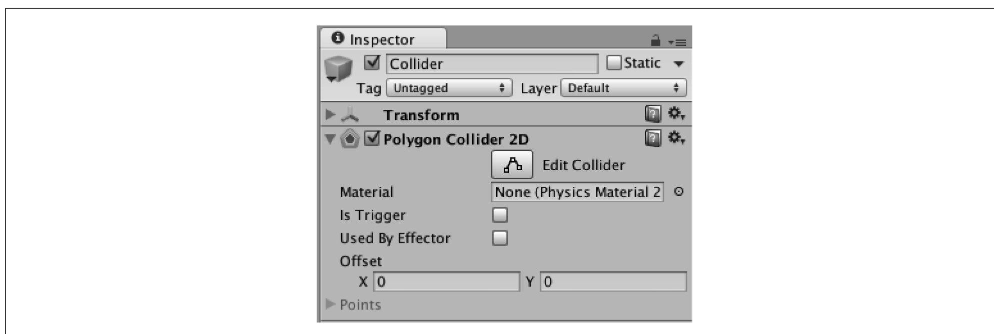


图 7-7：Edit Collider 按钮

在编辑模式下，可以将形状的各个点四处拖动，还可以单击并拖动连接每个点的线条来创建新点，或者在单击点的同时按下 Ctrl 键（Mac 上为 Command 键）来移除该点。

拖动各点，使它们大致匹配身体部位的形状，如图 7-8 所示。

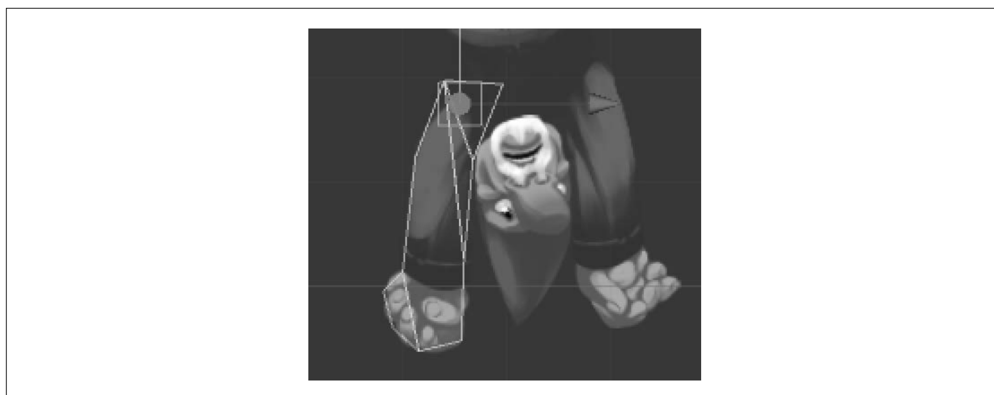


图 7-8：更新后的地精手臂的多边形碰撞器

完成之后，再次单击 Edit Collider 按钮。

刚刚添加的碰撞器应该大致如图 7-9 所示。

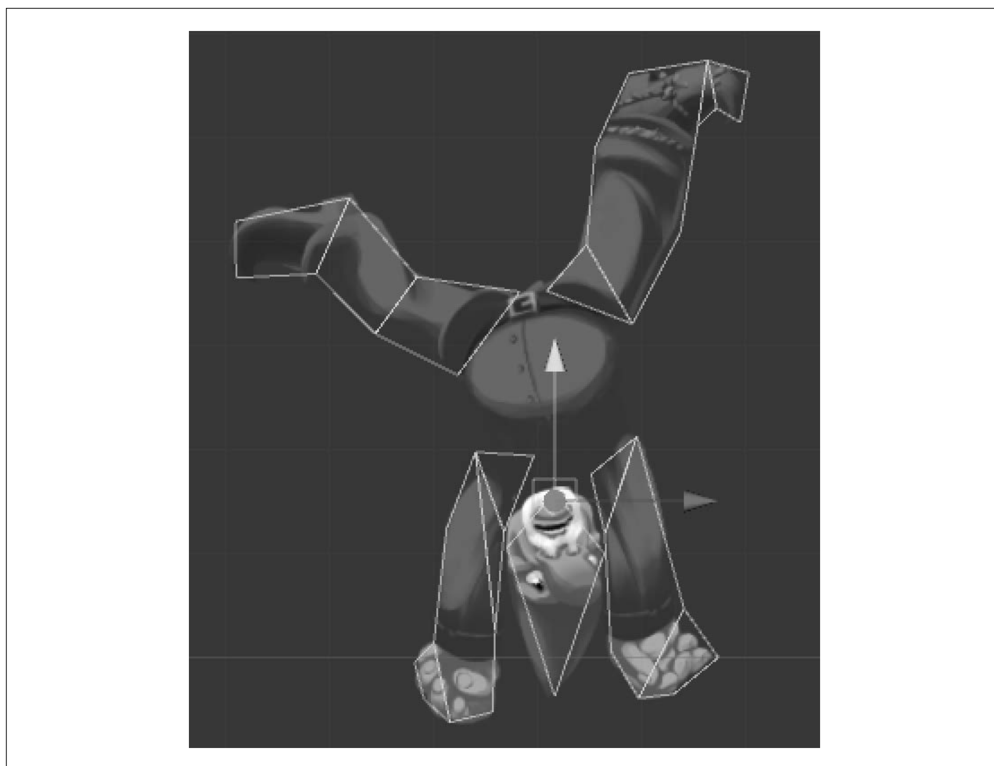


图 7-9：手臂、腿部和头部的碰撞器

还需要对碰撞器再做一处修改：需要稍微放大身体的圆形碰撞器，以匹配更加臃肿的躯干。

(3) **将 Body 的圆形碰撞器的半径增加到 1.2**。

这里想要实现的效果是让碰撞器大致匹配精灵的形状，但是不会重叠。这意味着在玩游戏期间，地精的身体部位不会以看起来很奇怪的方式彼此重叠。

现在碰撞器已经有了合适的形状，是时候更新关节了。回忆一下，头部、手臂和腿部都有铰链关节将其连接到躯干。我们需要确保轴点是正确的，以避免出现一些古怪的情况，例如地精的手臂看起来绕着上臂旋转。

(4) **更新地精关节的 Connected Anchor 和 Anchor 的位置**。对于除了躯干之外的身体部位，将 Connected Anchor 和 Anchor 拖动到轴点。腿部的轴点应该在臀部，手臂的轴点在肩部，头部的轴点在颈部。



如果把轴点和连接轴点拖动到靠近精灵的中心点位置，它们将被“吸附”到该中心点。

不要忘记，Leg Rope 上有两个关节：一个将其连接到躯干，另一个用于绳索。将第二个关节的 Anchor 移动到脚踝位置。

我们需要对地精的 `Gnome` 脚本做一些修改。还记得吗，前面提到过，当地精触碰到宝藏时，手臂精灵会发生变化？目前，这部分仍然在使用原来的原型图，与新的画面并不搭配。

(5) **更新 `Gnome` 脚本使用的精灵**。选择父 Gnome 对象。
将 ArmHoldEmpty 精灵拖动到地精的 Arm Holding Empty 框中，并将 ArmHoldFull 精灵拖动到地精的 Arm Holding Full 框中。

现在，当地精捡起宝藏时，手臂精灵将变为正确的图像。另外，当地精掉下宝藏时（地精因触碰到陷阱而死亡时，会发生这种情况），地精的手臂不会变回火柴人手臂。

最后，我们需要调整地精的大小，使其更适合游戏世界，然后把所做的修改保存到预设中。

(6) **调整地精的大小**。选择父 Gnome 对象，将 Scale 的 x 和 y 值从 0.5 改为 0.3。
(7) **把修改应用到预设**。选择父 Gnome 对象，单击 Inspector 顶部的 Apply 按钮。
(8) **从世界中移除地精**。保存之后就不需要在场景中保留地精了，所以我们删除它。

现在就完成了对地精的更新，接下来要更新 Game Manager，使其能够使用更新后的对象。

(9) **使 Game Manager 使用更新后的对象**。选择 Game Manager，将刚才更新的地精预设拖放到 Gnome Prefab 框中。
(10) **测试游戏**。现在更新后的地精就出现在了游戏世界中！图 7-10 显示了地精的样子。

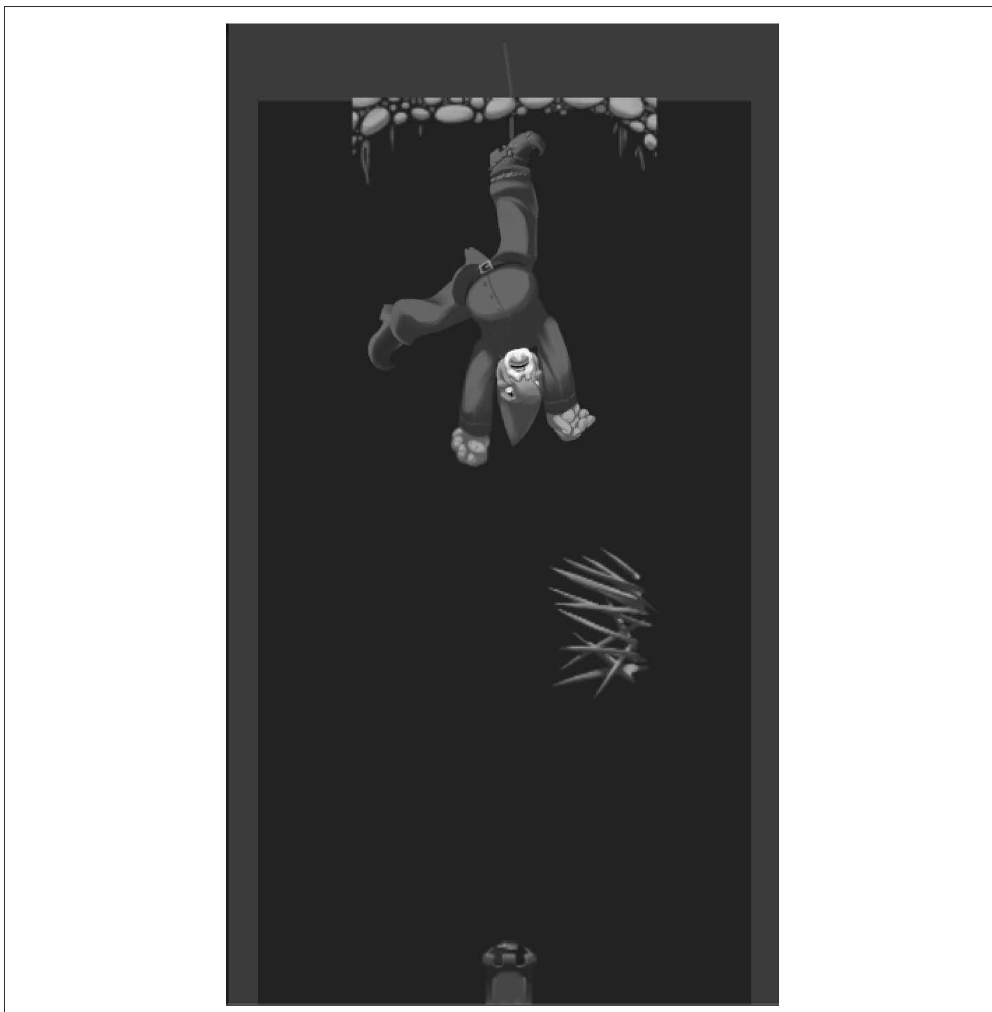


图 7-10：更新后的地精出现在游戏中

7.3　背景

目前的游戏背景是扁平的灰色四边形，看起来完全不像井的内部。下面我们就进行修改。

为了解决这个问题，我们将添加一组更加复杂的对象来表示背景和井壁。动手之前，确保已经把 Background 文件夹中的精灵添加到了你的项目中。

7.3.1　层

在添加图片之前，我们首先需要确定它们在场景中的顺序。开发 2D 游戏时，让正确的精灵显示在其他精灵之上是非常重要的，但有时候维护起来并不容易。幸好，Unity 内置的

一个解决方案——**排序层**——让这项工作简单了许多。

排序层是合起来绘制的一组对象。顾名思义，我们能够按照自己希望的顺序排列对象。这意味着我们能够把特定的对象分组到“背景”层，把其他对象分组到“前景”层，等等。另外，在每个层中可以进一步排序对象，这样就可以确保特定的背景部分总是在其他部分之后绘制。

在任何时候至少有一个排序层，即 Default（默认）。除非明确修改，否则所有新对象都将被放到这个层中。

我们将在这个项目中添加多个排序层。具体来说，我们将添加以下排序层。

- Level Background 层，其中包含关卡背景对象，总是显示在其他所有对象的后面。
- Level Foreground 层，其中包含前景对象，如井壁。
- Level Objects 层，其中包含陷阱等对象。

创建排序层需要执行下面的步骤。

(1) **打开 Tags & Layers Inspector**。打开 Edit 菜单，选择 Project Settings → Tags & Layers。

(2) **添加 Level Background 排序层**。打开 Sorting Layers，添加一个新层，命名为 Level Background。

将新层拖放到列表顶部（位于 Default 上方）。如此一来，这个层的任何对象都会显示在 Default 层的对象**后面**。

(3) **添加 Level Foreground 层**。重复上面的过程，添加一个名为 Level Foreground 的新层，并将其添加到 Default 层的**下方**。于是，这个层中的任何对象都会显示在 Default 层中的对象**前面**。

(4) **添加 Level Objects 层**。最后，再次重复上面的过程，添加一个名为 Level Objects 的新层，并将其置于 Default 层的下方和 Level Foreground 层的上方。这是绘制陷阱和宝藏的层，也就是说，它们需要位于前景之后。

7.3.2 创建背景

设置好层之后，就可以开始构建背景了。背景有 3 个不同的主题，即棕色、蓝色和红色，每个主题都包含多个精灵：背景、井壁精灵，以及井壁精灵的背景版本。

因为我们想让关卡内容的布局符合自己的品味，所以最好为每个主题都创建预设。我们首先创建 Brown 背景主题，构建对象，并将其保存为预设。然后，对 Blue 和 Red 主题重复相同的过程。

然而，在开始**这个过程**之前，为了看上去整洁，我们需要创建一个对象来包含全部关卡背景对象。需要执行的步骤如下所示。

(1) **创建 Level 容器对象**。打开 GameObject 菜单，选择 Create Empty，创建一个新的空游戏对象，命名为 Level，并设其位置为 (0,0,1)。

(2) **创建 Background Brown 对象的容器**。创建另外一个游戏对象，命名为 Background Brown。使该对象成为 Level 对象的子对象，并确保其位置为 (0,0,0)。这样该对象就不会偏离 Level 对象的位置。

(3) **添加主背景精灵**。将 BrownBack 精灵拖放到场景中，使其成为 Background Brown 对象的子对象。

选择这个新精灵，将其 Sorting Layer 改为 Level Background。最后，将其 *x* 位置设为 0，使其居中显示。

(4) **添加背景井壁对象**。将 BrownBackSide 精灵拖放到场景中，使其成为 Background Brown 对象的子对象。

将其 Sortling Layer 设为 Level Background，Order in Layer 设为 1。这将使该对象显示在主背景之前，同时仍然位于其他层所有对象的后面。

将其 *x* 位置设为 −3，使其被推到左侧。

(5) **添加前景井壁对象**。将 BrownSide 精灵拖动到场景中，使其成为 Background Brown 精灵的子对象。将 Sorting Layer 设为 Level Foreground。

将其 *x* 位置设为 −3.7，*y* 位置设为与 BrownBackSide 精灵相同。我们想让它们水平对齐，但是前景对象稍微往左一些。

因为井壁对象的高度只有主背景图片的一半，所以我们再创建一行井壁对象。选择 BrownBackSide 和 BrownSide 精灵，然后按 Ctrl-D（Mac 上为 Command-D）键复制井壁对象。

向下移动新的井壁对象，使上一行的底边与下一行的顶边重合。完成之后，背景应该如图 7-11 所示。

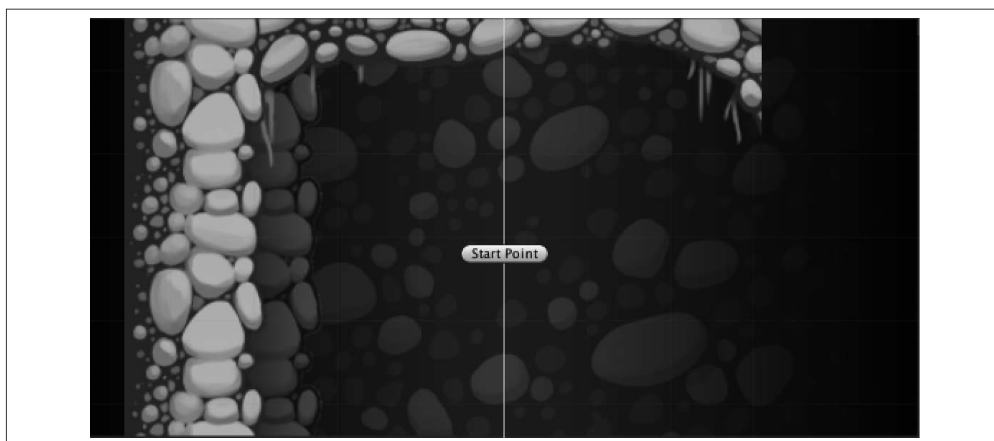


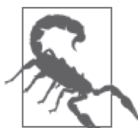
图 7-11：部分更新的背景

现在设置好了左侧的井壁对象，接下来该设置右侧了。为此，我们将复制现有的精灵并进行调整，使它们适合右侧。

(1) **再次复制井壁对象**。选择全部 BrownSide 和 BrownBackSide 对象，按 Ctrl-D（Mac 上为 Command-D）键。

(2) **确保将 Pivot Mode 设为 Center**。如果 Pivot Mode 按钮被设为 Pivot，则单击该按钮，使其变为 Center。

(3) **旋转对象**。使用 Rotate 工具，将右侧对象旋转 180°。按住 Ctrl 键（Mac 上的 Command 键）使旋转更加准确。



不要使用 Inspector 修改旋转值，因为这会让对象围绕各自的原点进行旋转。我们想要实现的是让对象围绕公共的中心点旋转。

(4) **垂直翻转对象**。将对象 Scale 的 y 值改为 −1。如果不执行此操作，光照效果在上下颠倒的时候看起来会不正确。

完成上述操作之后，这些对象的 Transform Inspector 应该如图 7-12 所示。

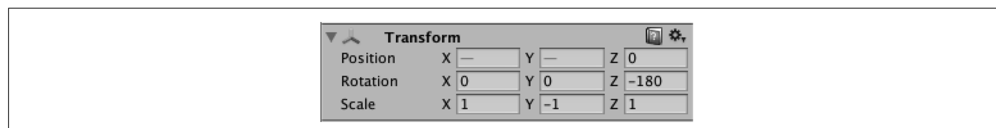


图 7-12：右侧背景元素的 Transform 组件

(5) **将新对象移动到关卡的右侧**。毕竟，它们属于这个位置。完成后的效果应该如图 7-13 所示。

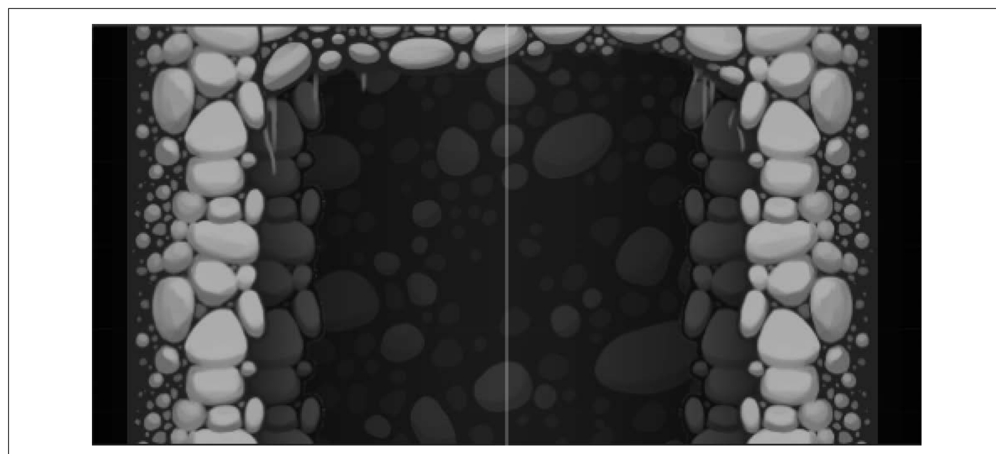


图 7-13：更新后的背景

设置好 Background Brown 对象之后，需要将其转换成预设。执行下面的步骤。

(1) **从 Background Brown 对象创建一个预设**。将 Background Brown 对象拖放到 Project 选项卡，将创建一个预设。将此预设移动到 Level 文件夹。

(2) **复制 Background Brown 对象**。选择 Background Brown 对象，按几次 Ctrl-D（Mac 上为 Command-D）键。向下移动每个新对象，直到有了一个长度合适的背景区。

7.3.3　不同的背景

创建好第一个背景后，可以按照相同的步骤创建其他两个主题的背景。

(1) **创建 Background Blue 主题**。创建一个新的空对象，命名为 Background Blue，使其成为 Level 对象的子对象。

按照与创建 Background Brown 对象相同的步骤，只不过使用的是 BlueBack、BlueBackSide 和 BlueSide 精灵。

不要忘记，在创建完成之后，从 Background Blue 对象生成一个预设。

(2) **创建 Background Red 主题**。同样，执行相同的步骤，只不过使用的是 RedBack、RedBackSide 和 RedSide 精灵。

完成之后，关卡应该如图 7-14 所示。



图 7-14：背景区

这种设置有一个问题：颜色一致的背景对象衔接得很好，但是不同颜色的对象衔接处会出现突兀的接缝。

为了解决这个问题，我们将叠加精灵，把接缝遮住。这些精灵将放到 Level Foreground 层中，并显示在游戏中其他所有对象的前方。

(1) **添加 BlueBarrier 精灵**。此精灵用于遮盖 Brown 背景与 Blue 背景之间的接缝。将其放到 Brown 背景和 Blue 背景交界处，并设为 Level 对象的子对象。

(2) **添加 RedBarrier 精灵**。此精灵用于遮盖 Blue 背景与 Red 背景之间的接缝。将其放到 Blue 背景和 Red 背景交界处，并设为 Level 对象的子对象。

(3) **更新这两个精灵的排序层**。选择 BlueBarrier 和 RedBarrier 精灵，将它们的 Sorting Layer 设为 Level Foreground。

接下来，将 Order in Layer 设为 1，使叠加精灵显示在井壁的前方。

完成之后，关卡应该如图 7-15 所示。

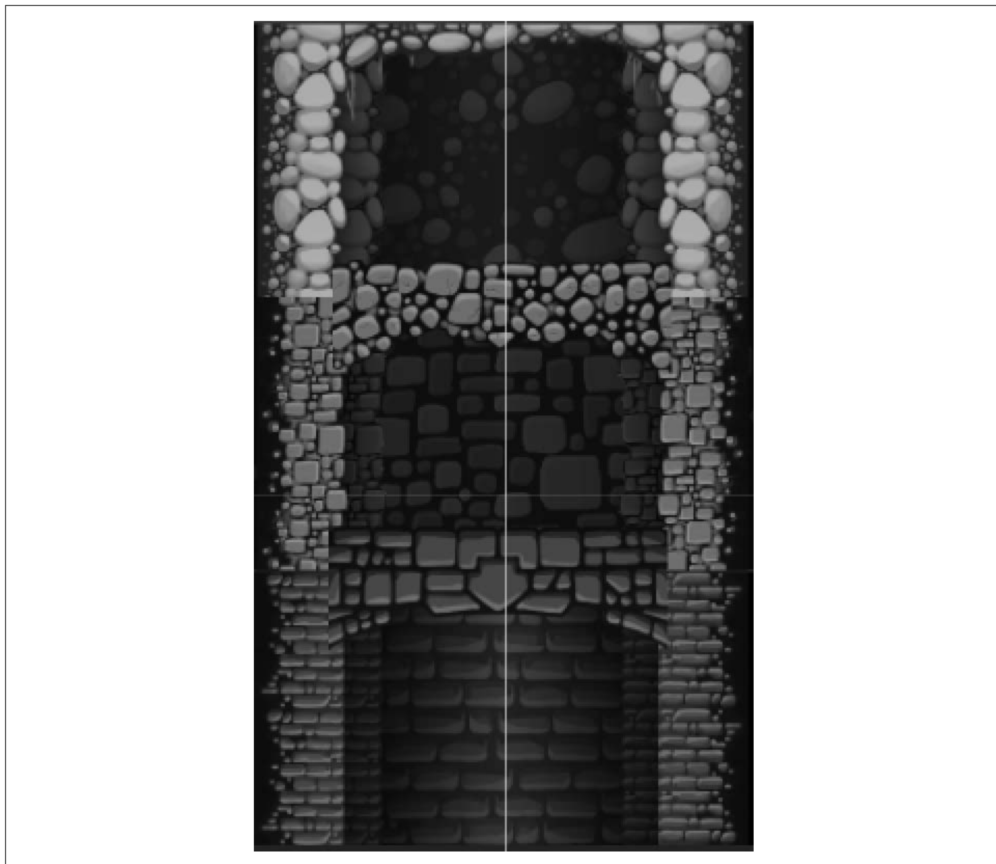


图 7-15：添加了 Barrier 精灵后的背景

7.3.4 井底

最后，还需要添加井底。在这个游戏中，井底是干燥的，并且沙层覆盖了井的最底层，还有一些沙土也附着到了井壁上。在场景中添加这个效果，要执行下面的步骤。

(1) **为井底精灵创建容器对象**。创建一个新的空游戏对象，命名为 Well Bottom，设为 Level 对象的子对象。

(2) **添加井底精灵**。将 Bottom 精灵拖放到场景中，并添加为 Well Bottom 对象的子对象。

将 Sorting Layer 设为 Level Background，Order in Layer 设为 2。这将使其位于 Background 和 Background Side 精灵的前方，但是位于游戏中其他对象的后方。

将精灵放到井底，设其 x 位置为 0，使其与关卡的其余精灵对齐。

(3) **在井的左侧添加井壁装饰精灵**。将 SandySide 精灵拖放到场景中，添加为 Well Bottom 对象的子对象。

将 Sorting Layer 设为 Level Foreground，Order in Layer 设为 1，使其显示在井壁的前方。接下来，将精灵移动到左侧，使其与井壁对齐（效果应该如图 7-16 所示）。

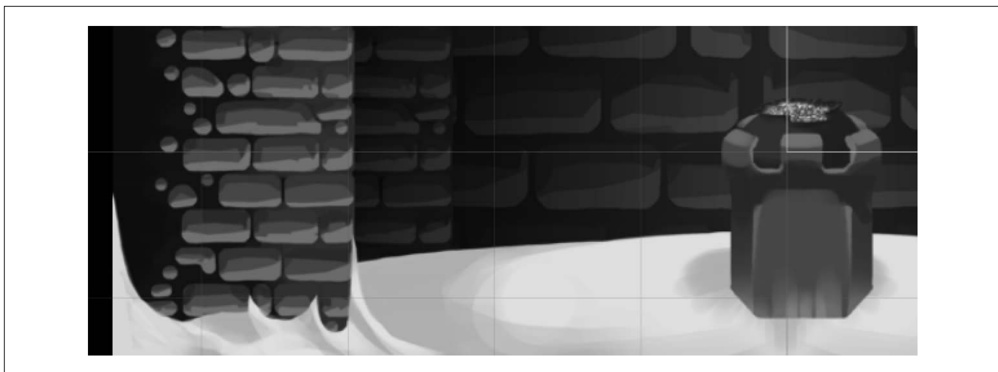


图 7-16：SandySide 精灵与井底对齐

(4) **添加右侧的井壁对象**。复制 SandySide 精灵，将其 Scale 的 x 值设为 −1 以进行水平翻转，然后移动到井的右侧。

(5) **确保宝藏位于正确的位置**。调整宝藏精灵的位置，使其位于沙层的中间位置。

完成之后的效果应该如图 7-17 所示。

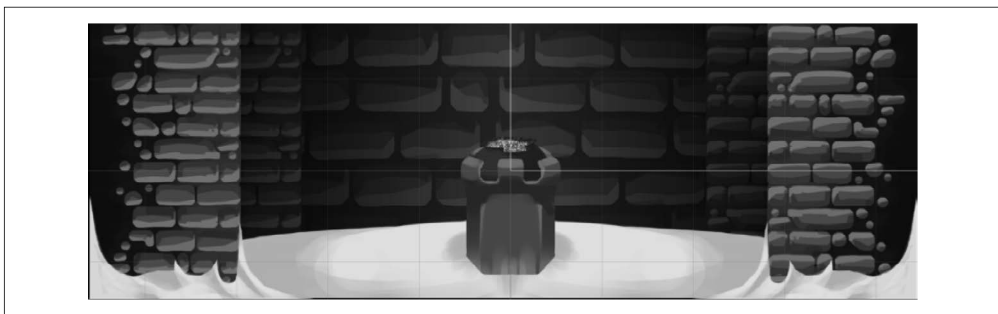


图 7-17：完成后的井底

7.3.5 更新摄像机

为了使新背景适合游戏，还有最后一项工作要做：更新摄像机。需要修改的地方有两个：首先，更新摄像机，使玩家能够看到整个关卡；其次，考虑到更新后关卡的大小，包含摄像机位置的脚本也需要更新。执行以下步骤来配置摄像机。

(1) **更新摄像机的大小**。选择 Main Camera 对象，将摄像机的 Ortho Size 设为 7。这将给玩家足够宽的视野，以便看到整个关卡。

(2) **更新摄像机的限高**。我们已经修改了摄像机能够看到的范围，还需要调整摄像机的限高。将摄像机的 Top Limit 改为 11.5。

我们还需要调整 Bottom Limit，但是这里选择的值取决于井有多深。

确定这个值最好的方法是把地精下降到最低的地方。如果在到达井底之前，摄像机就停止移动，则减小 Bottom Limit；如果摄像机超出了井底（显示了蓝色的背景），则增大 Bottom Limit。

在停止游戏之前，要记下这个值，因为游戏结束后会重设为原始值。停止游戏后，把刚才记下的数字输入 Bottom Limit 字段中。

7.4 用户界面

现在是时候改进游戏的 UI 了。前面在设置界面时，使用了 Unity 提供的标准按钮。虽然它们能够提供必要的功能，但是不太符合这个游戏的观感，因此需要用更好的东西来替换按钮图片。

另外，当地精到达井口时，需要显示一个 Game Over 画面；当玩家暂停游戏时，显示另外一个画面。

在继续设置之前，确保导入了本节需要用到的精灵。导入精灵的 Interface 文件夹，将其添加到 Sprites 文件夹中。

这些精灵被设计成高分辨率的图片，以便用在各种不同的场景中。为了使其在游戏中能用作按钮，将其添加到 Canvas 中时，Unity 需要知道它们应该是什么尺寸。这可以通过调整这些精灵的 Pixels Per Unit 值做到，当把精灵添加到 UI 组件或者精灵渲染器时，该值控制着它们的比例。

选择 Interface 文件夹中的全部图片（但不要选择 You-Win）以配置精灵。将 Pixels Per Unit 值改为 2500。

首先处理当前位于窗口右下角的 Up 和 Down 按钮，将其更新为更美观的图片。为此，需要移除按钮上的标签，并调整每个按钮的大小和位置，以适应它们的新图片。需要执行的步骤如下所示：

(1) **移除 Down 按钮的标签**。找到 Down Button 对象，移除其 Text 子对象。

(2) **更新精灵**。选择 Down Button 对象，将 Source Image 属性改为 Down 精灵（位于 Interface 文件夹中）。

单击 Set Native Size 按钮，按钮会调整自己的大小。

最后，调整按钮的位置，使其仍然位于画面的右下角。

(3) **更新 Up 按钮**。为 Up 按钮重复相同的过程。移除 Text 子对象，并将 Source Image 改为 Up 精灵。接下来，单击 Set Native Size 按钮，并更新其位置，使其位于 Down 按钮的上方。

(4) **测试游戏**。按钮仍然能够工作，看起来效果也更好了（如图 7-18 所示）。

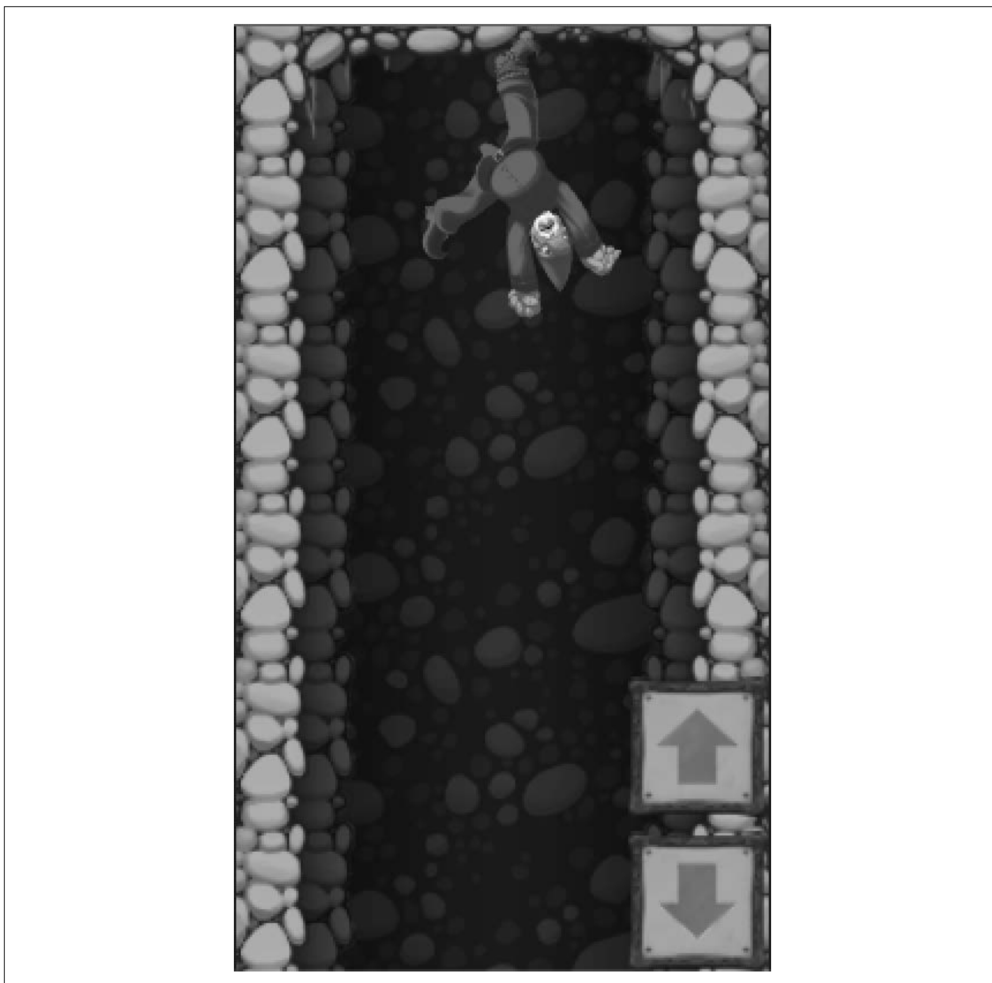


图 7-18：更新后的 Up 和 Down 按钮

现在把这些按钮分组到一个容器中。这么做有两个原因：首先，让 UI 整洁有序是个好主意；其次，把按钮分组到一个对象中，能够同时启用和禁用它们。后面章节中实现 Pause 菜单时，这会很有用。执行以下步骤来进行设置。

(1) **创建按钮的父对象**。创建一个新的空游戏对象，命名为 Gameplay Menu，设为 Canvas 的子对象。

(2) **设置对象，使其填充整个画面**。将 Gameplay Menu 的锚点设为水平和垂直拉伸。单击左上角附近的 Anchors，然后单击弹出菜单右下角的选项（如图 7-19 所示）。

然后，将 Left、Top、Right 和 Bottom 设为 0。这将使整个对象填充其整个父对象（即 Canvas，这样整个对象就将填充整个 Canvas）。

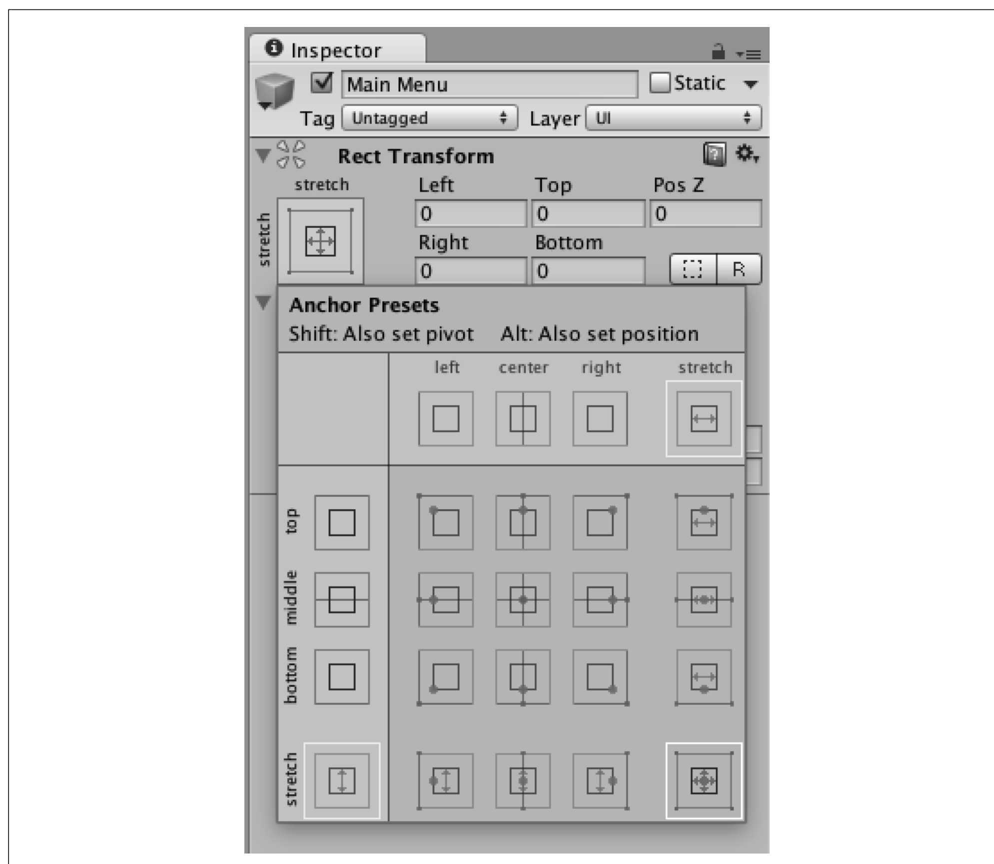


图 7-19：将对象的锚点设为水平和垂直拉伸

(3) **将按钮移动到 Gameplay Menu 对象中**。将 Up 按钮和 Down 按钮在 Hierarchy 中的条目拖放到 Gameplay Menu 对象中。

接下来将创建 You Win 画面。该画面向玩家显示一个图片，以及一个让玩家再次玩游戏的按钮。为了准备此画面，执行下面的步骤。

(1) **为 Game Over 画面创建容器对象**。创建一个新的空游戏对象，命名为 Game Over，设为 Canvas 的子对象。

按照设置 Gameplay Menu 对象的步骤，使其水平和垂直拉伸。

(2) **添加 Game Over 图片**。通过打开 GameObject 菜单，选择 UI → Image，创建一个新的 Image 游戏对象，设为刚才创建的 Game Over 对象的子对象。

将新的 Image 对象的锚点设为水平和垂直拉伸。将 Left 和 Right 边距设为 30，将 Bottom 边距设为 60。这将在图片周边添加一些填充，并确保它不会遮盖我们将要添加的 New Game 按钮。

将 Image 的 Source Image 属性设为 You Win 精灵，并打开 Preserve Aspect 选项以防止其被拉伸。

(3) **添加 New Game 按钮**。打开 GameObject 菜单并选择 UI → Button，为 Game Over 对象添加一个新的 Button。

将新按钮的标签文本设为 New Game，锚点设为底部居中。

将按钮移动到画面底部居中的位置。完成之后，界面应该如图 7-20 所示。



图 7-20：Game Over 界面

(4) **将 New Game 按钮连接到 Game Manager**。单击该按钮时，我们想让 Game Manager 重置游戏。这可以通过调用 `GameManager` 脚本的 `RestartGame` 函数实现。

单击 Button 的 Inspector 底部的 + 按钮，将 Game Manager 拖放到出现的框中。接下来，将函数改为 `GameManager` → `RestartGame`。

现在需要把 Game Manager 连接到这些新的 UI 元素。`GameManager` 脚本已设置好，能够根据游戏的状态启用和禁用合适的用户界面元素：当游戏正在进行时，脚本会试图激活 Gameplay Menu 变量中的对象，并禁用其他菜单。执行下面的步骤来进行配置和测试。

(1) **将 Game Manager 连接到菜单**。选择 Game Manager，将 Gameplay Menu 对象拖放到 Gameplay Menu 框中。接下来，将 Game Over 对象拖动到 Game Over Menu 框中。

(2) **测试游戏**。将地精一直坠到井底，捡起宝藏，然后到达出口。Game Over 画面将会显示。

我们还需要设置最后一个菜单，即 Pause 菜单，以及用于暂停游戏的按钮。Pause 按钮将显示在画面的右上角，当玩家触摸该按钮时，游戏将暂停，并显示恢复游戏和重新启动游戏的按钮。

为了设置暂停按钮，创建一个新的 Button 对象，将其命名为 Menu Button，设为 Gameplay Menu 对象的子对象。

- 移除 Text 子对象，将按钮的 Source Image 设为 Menu 精灵。
- 单击 Set Native Size 按钮，将按钮移动到画面的右上角。将锚点设为右上角。
- 完成之后，新按钮应该如图 7-21 所示。

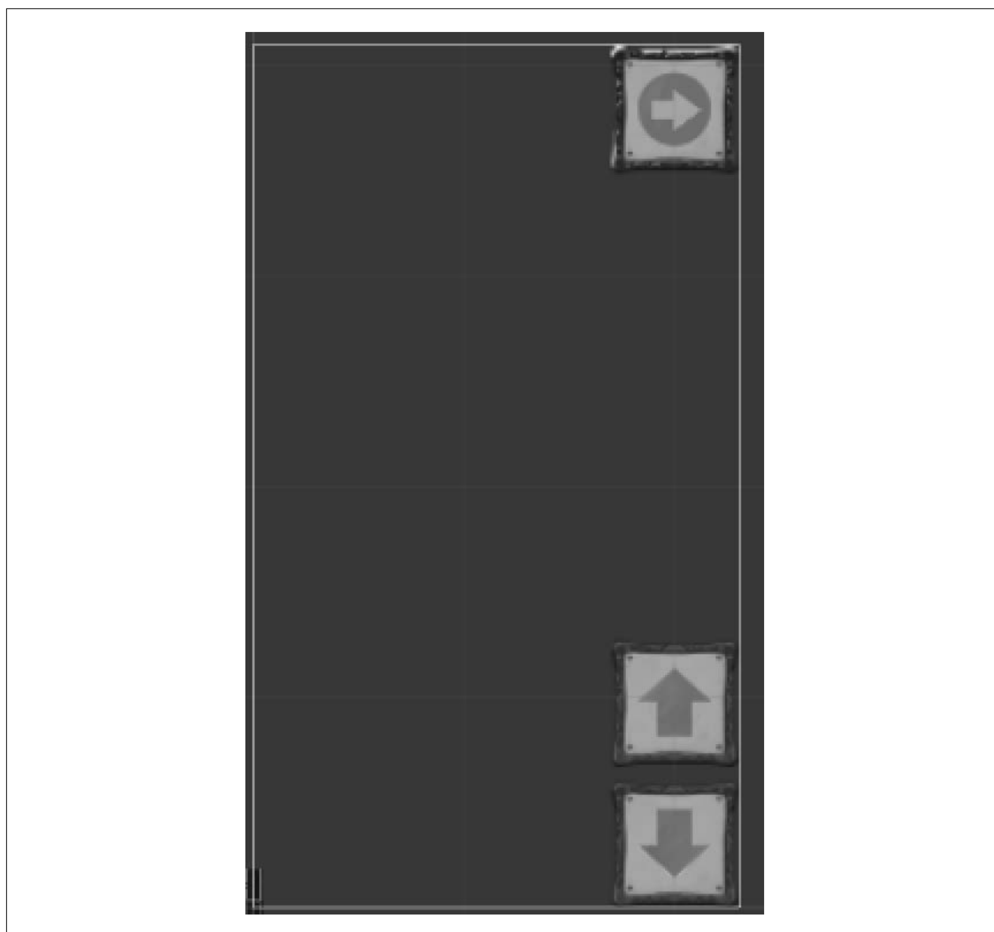


图 7-21：Menu 按钮

接下来，需要把这个按钮连接到 Game Manager。触摸此按钮时，将告诉 Game Manager 进入 Paused 状态。这将显示 Pause Menu（我们稍后创建），隐藏 Gameplay Menu，并暂停游戏。

为了把 Menu 按钮连接到 Game Manager，单击按钮的 Inspector 底部的 + 按钮，将 Game Manager 拖放到出现的框中。

使按钮调用 `GameManager.SetPaused`。选中复选框，使得当触摸按钮时，向 `SetPaused` 按钮发送 `true` 参数。

现在可以设置暂停游戏时显示的菜单。

(1) **创建 Main Menu 容器**。创建一个新的空对象，命名为 Main Menu，设为 Canvas 的子对象，将其锚点设为水平和垂直拉伸。其 Left、Right、Top 和 Bottom 边距设为 0。

(2) **将按钮添加到 Main Menu**。添加两个按钮，命名为 Restart 和 Resume。使这两个按钮成为刚才创建的 Main Menu 对象的子对象，并将各自的标签文本更新为 Restart Game 和 Resume Game。

完成之后，Main Menu 应该如图 7-22 所示。



图 7-22：Main Menu

(3) **将按钮连接到 Game Manager**。选择 Restart 按钮，使其调用 Game Manager 对象的 `GameManager.RestartGame` 函数。

接下来，选择 Resume 按钮，使其调用 Game Manager 的 `GameManager.Reset` 函数。

(4) **将 Main Menu 连接到 Game Manager**。Game Manager 需要知道调用 SetPaused 函数时应该显示哪个对象。选择 Game Manager，将 Main Menu 对象拖放到 Game Manager 的 Main Menu 框中。

(5) **测试游戏**。现在可以暂停并恢复游戏。另外，还可以重新启动整个游戏。

7.5 无敌模式

电子游戏中出现作弊码，其实源于一个非常实际的需求。构建游戏时，必须击败游戏中包含的各种陷阱和谜题来到达想要测试的特定部分，这个过程可能变得非常枯燥。为了加快开发进度，常见的做法是添加工具来修改游戏的玩法：射击游戏常常包含一些作弊码，让敌人不攻击玩家；策略游戏则可以添加作弊码来禁用战争迷雾。

我们的游戏也不例外：在构建游戏时，不应该每次运行游戏都必须应对每个障碍。为此，我们将添加一个工具来使地精变得无敌。

此功能将实现为一个复选框（有时叫作**开关**，toggle），显示在画面的左上角。开启该选项后，地精不会死亡。它仍然会受到伤害，显示下一章将添加的粒子效果，这对于测试很有用。

为了保持整洁有序，我们将把此复选框包含在一个容器对象中，就像其他 UI 组件一样。首先创建这个容器。

(1) **创建 Debug Menu 容器**。创建一个新的空游戏对象，命名为 Debug Menu，设为 Canvas 的子对象。将其锚点设为水平和垂直拉伸，并将 Left、Right、Top 和 Bottom 边距设为 0，使其填满画面。

(2) **添加 Invincible（无敌模式）开关**。打开 GameObject 菜单并选择 UI → Toggle，创建一个新的 Toggle 对象，命名为 Invincible。

将新对象的锚点设为 Top Left，并将其移动到画面的左上角。

(3) **配置开关**。选择 Lable 对象，这是刚才添加的开关的子对象，并将其 Text 组件的颜色设为白色。将标签的文本设为 Invincible。

将 Toggle 对象的 Is On 属性设为关闭。

完成之后，开关应该如图 7-23 所示。

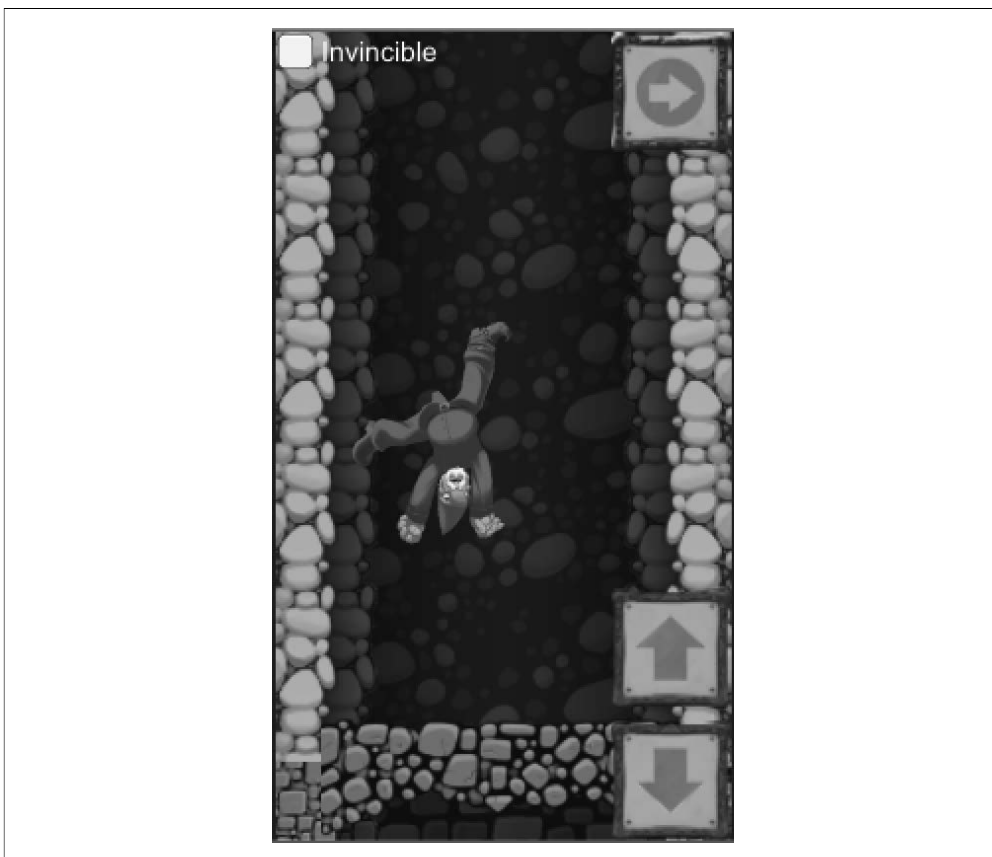


图 7-23：显示在画面左上角的 Invincible 复选框

(4) **将开关连接到 Game Manager**。通过单击 + 按钮，在 Invincible 开关的 Value Changed 事件中添加一个新条目。将 Game Manager 拖放到显示的框中，并将函数修改为 `GameManager.gnomeInvincible`。现在，当开关的值改变时，`gnomeInvincible` 属性也会改变。

(5) **测试游戏**。玩游戏，并开启 Invincible。现在，即使地精触碰到陷阱也不会死亡。

7.6　小结

现在，游戏看上去很不错。核心的游戏玩法已经完成，感觉很好，我们也添加了一些开发工具，让测试游戏变得更加容易。不过，我们还可以锦上添花。下一章中，我们将添加更多内容，进行更多优化，并通过构建菜单结构和音效，最终完成这个游戏的开发。

第8章

完成*Gnome's Well*游戏

8.1 更多陷阱和关卡对象

游戏开始成型：我们已经更新了地精的美术作品和游戏的 UI，背景看上去也不错。目前，我们只有一种类型的陷阱：棕色的尖刺。接下来，我们将为这些静态的尖刺创建两种主题化版本，让陷阱更加多样。

我们还将添加一种新的陷阱类型：转轮。转轮会造成与尖刺相同的伤害，但是更复杂一些：它由 3 个精灵组成，其中一个是动画。

最后，我们将添加一些不会造成伤害的东西，即井壁，以及玩家需要绕过的障碍。当与陷阱结合起来时，这些对象迫使玩家仔细思考如何在关卡中前进。

8.1.1 尖刺

我们首先创建主题尖刺。对于现有的精灵，我们已经有了预设，所以只需要更新精灵，并重新生成碰撞器。这些通过执行下面的步骤来实现。

(1) **为尖刺创建新的预设**。选择 SpikesBrown 精灵，按 Ctrl-D（Mac 上为 Command-D）键创建一个副本。将新对象命名为 SpikesBlue。
创建另外一个副本，命名为 SpikesRed。

(2) **更新精灵**。选择 SpikesBlue 预设，将精灵改为 SpikesBlue 图片。

(3) **更新多边形碰撞器**。因为多边形碰撞器与 Sprite Renderer 位于相同的对象上，所以碰撞器使用精灵来计算其形状。但是，当精灵变化时，碰撞器不会自动更新形状。为了解决这个问题，需要重置多边形碰撞器。
单击 Polygon Collider 2D 组件右上角的齿轮图标，然后在显示的菜单中单击 Reset 按钮。

(4) **更新 SpikesRed 对象**。现在我们已经完成了 SpikesBlue 对象的设置，只需要对 SpikesRed 执行相同的步骤即可（但是要使用 SpikesRed 图片）。

完成上述操作后，关卡中就添加了一些 SpikesBlue 和 SpikesRed 对象。

8.1.2 转轮

接下来添加转轮。转轮在游戏中比尖刺伸出得更远，并有一个令人胆战心惊的旋转刀轮。就底层逻辑而言，转轮与尖刺实际上是相同的：地精触碰到它时就会死亡。不过，在游戏中添加各种不同的陷阱，有助于多样化关卡，从而保持玩家的兴趣。

由于转轮有动画效果，需要使用多个精灵进行设置。另外，我们将把其中一个精灵——刀轮——设为高速旋转。

为了构建转轮，将 SpinnerArm 精灵拖动到场景中，并将其 Sorting Layer 设为 Level Objects。

将 SpinnerBladesClean 精灵拖出，作为子对象添加到 SpinnerArm 对象。将其 Sorting Layer 设为 Level Objects，Order in Layer 设为 1。将其置于转轮臂的顶部，*x* 位置设为 0，使其准确居中。

将 SpinnerHubcab 精灵拖出，作为子对象添加到 SpinnerArm 对象。将其 Sorting Layer 设为 Level Objects，Order in Layer 设为 2，*x* 位置设为 0。完成之后，转轮应该如图 8-1 所示。



图 8-1：构造好的转轮

现在添加一个 `SignalOnTouch` 脚本，让转轮对地精造成伤害。当地精触碰到转轮所附加的碰撞器时，`SignalOnTouch` 脚本将发送一个消息。为了使其能够工作，我们还需要添加一个碰撞器。执行下面的设置步骤。

(1) **为刀轮添加一个碰撞器**。选择 SpinnerBladesClean 对象，添加一个 Circle Collider 2D。将其半径减为 2，这将减小碰撞盒的尺寸，便于处理转轮。

(2) **添加 Signal On Touch 组件**。单击 Add Component 按钮，添加一个 `SignalOnTouch` 脚本。单击 Inspector 底部的 + 按钮，将 Game Manager 拖放到框中。将函数改为 `GameManager.TrapTouched`。

接下来就让刀轮旋转起来。为此，我们将添加一个 Animator 对象，将其配置为运行一个 Animation。这个 Animation 很简单，只需要一圈圈转动自己附加到的对象。

设置 Animator 需要创建一个 Animator Controller。Animator Controller 允许使用参数定义当前播放的 Animation。我们不会在这个游戏中使用 Animator Controller 的高级功能，但是了解这些高级功能会很有帮助。设置 Animator 的步骤如下。

(1) **添加 Animator**。选择刀轮，并添加一个新的 Animator 组件。

(2) **创建 Animator Controller**。在 Level 文件夹中，创建一个新的 Animator Controller 资源，命名为 Spinner。
在 Level 文件夹中，创建一个新的 Animation 资源，命名为 Spinning。

(3) **让 Animator 使用新的 Animator Controller**。选择刀轮，将刚才创建的 Animator Controller 拖放到 Controller 框中。

接下来设置 Animator Controller。

(1) **打开 Animator**。双击 Animator Controller，Animation 选项卡将会打开。

(2) **把 Spinning 动画添加到 Animator Controller 中**。将 Spinning 动画拖放到 Animator 窗格。Animator Controller 中现在应该只有一个动画状态，以及预先存在的 Entry、Exit 和 Any State 项，如图 8-2 所示。

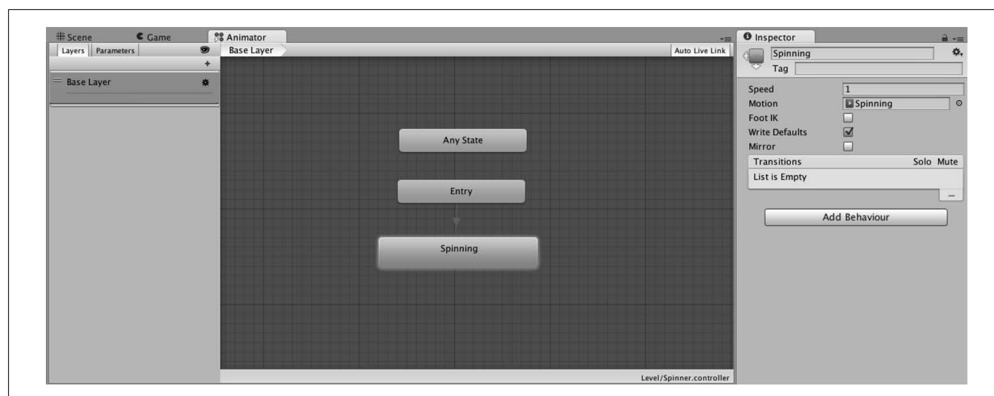


图 8-2：Spinner 的 Animator Controller

现在已经把 Animator 设置为使用 Animator Controller，Animator Controller 则设置为开始播放 Spinning 动画。接下来就设置这个动画，使其产生旋转效果。

(1) **确保选中转轮的刀轮**。回到场景视图，再次选择刀轮。

(2) **打开 Animation 窗格**。打开 Window 菜单，选择 Animation，将会打开 Animation 选项卡。将该选项卡拖放到一个方便操作的地方。还可以单击该窗格顶部的选项卡，在 Unity 主窗口中四处拖动，将其停靠在 Unity 的另外一个部分。

   继续操作之前，确保在 Animation 窗格的左上角选择了 Spinning 动画。

(3) **为转轮的 Rotation 属性添加一个曲线**。单击 Add Property 按钮，将出现一个可显示动画的组件列表。找到 Transform → Rotation 元素，然后单击列表右侧的 + 按钮。

默认情况下，新属性有两个关键帧：一个位于动画开始位置，另一个位于动画结束位置，如图 8-3 所示。

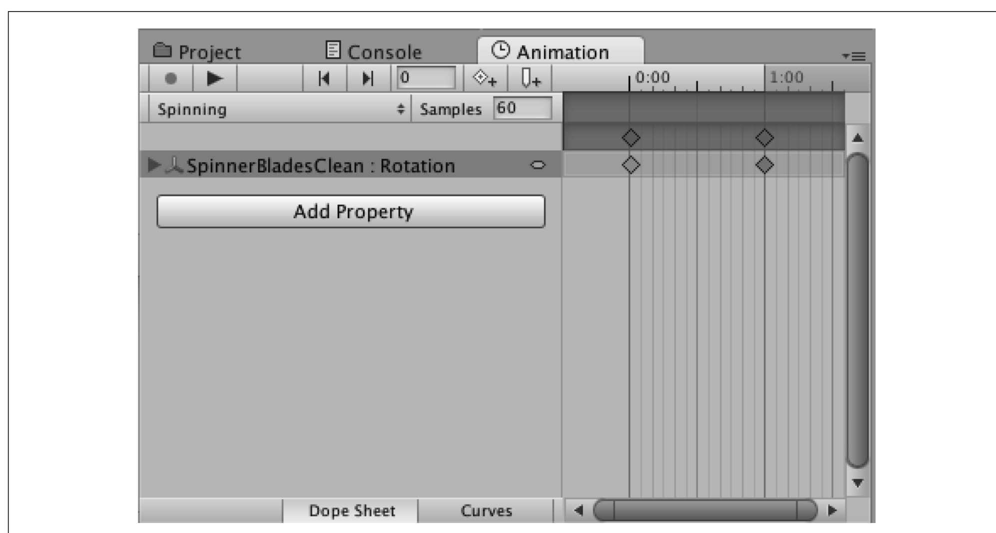


图 8-3：新建动画的关键帧

我们想让对象旋转 360°。这意味着对象在动画开始时旋转 0°，在动画结束时旋转 360°。为了实现这个效果，需要修改动画的最后一个关键帧。

(1) **选择最右侧的关键帧**。

(2) 单击 Animation 窗格最右侧的菱形，动画将跳到时间轴的该点。Unity 现在将进入**记录模式**，意味着对转轮做出的修改将被记录下来。还可以注意到，Unity 窗口顶部的控件变成了红色，以提醒你这个事实。

   查看 Inspector 中的 Transform 组件，可注意到 Rotation 值也是红色的。

(3) **更新旋转**。将 Rotation 的 *z* 值改为 360。

(4) **测试动画**。单击 Animation 选项卡的 Play 按钮，观看刀轮旋转。如果转得不够快，则单击并拖动结束位置的关键帧，使其靠近开始的关键帧。这能够减少动画时长，使对象更快完成旋转。

(5) **使动画循环播放**。进入 Project 窗格，选择你创建的 Spinning 动画资源。在 Inspector 中，确保选中 Loop Time 复选框。

(6) **玩游戏**。刀轮现在将旋转。

在转轮投入使用之前，还需要做最后一项设置：把转轮缩小，使其匹配游戏的其余部分。

(1) **缩放转轮**。选择父对象 SpinnerArm，将 Scale 的 x 值和 y 值设为 0.4。

(2) **将转轮转换为预设**。将 SpinnerArm 对象拖放到 Project 窗格中。这将创建一个名为 SpinnerArm 的新预设，将其重命名为 Spinner。

现在就可以旋转转轮，将其放到关卡中。地精触碰到转轮时将会死亡。

8.1.3　障碍

除了陷阱以外，添加一些**不会**让地精死亡的障碍是一个好主意。这些障碍能够让玩家放慢速度，并迫使他们思考如何绕过你添加的各种陷阱。

这些障碍属于你在游戏中添加的最简单的对象：只由一个精灵渲染器和一个碰撞器组成。它们非常简单，且彼此相似，因此可以为它们同时创建预设。设置障碍的步骤如下。

(1) **将障碍精灵拖出**。在场景中添加 BlockSquareBlue、BlockSquareRed 和 BlockSquareBrown 精灵。接下来，在场景中添加 BlockLongBlue、BlockLongRed 和 BlockLongBrown 精灵。

(2) **添加碰撞器**。选择全部 6 个对象，并单击 Inspector 底部的 Add Component 按钮。添加一个 Box Collider 2D 组件，每个障碍将得到一个绿色的盒形碰撞器。

(3) **将其转换为预设**。将每个障碍拖放到 Level 文件夹以创建预设。

现在已经完成了障碍的设置，可以在关卡中添加障碍和井壁了。这项工作很容易。

8.2　粒子效果

地精死亡时，简单地让其肢体断开，并不是特别令人满意的视觉效果。为了创建更有趣的效果，我们将添加粒子系统。

具体来说，我们将添加一种当地精触碰到陷阱时显示的粒子效果（“血爆”），以及一种当地精的肢体断开时显示的粒子效果（“血流”）。

8.2.1　定义粒子的材质

因为上述两个粒子系统将发射相同的东西（地精的血），所以先创建这两个粒子系统都会使用的材质。执行下面的步骤来创建和准备材质。

(1) **配置 Blood 纹理**。找到并选中 Blood 纹理，将其类型从 Sprite 改为 Default，并确保选中 Alpha Is Transparency（如图 8-4 所示）。

(2) **创建 Blood 材质**。打开 Asset 菜单，选择 Create → Material，创建一个新的 Material 资源，命名为 Blood，并将其 Shader 改为 Unlit → Transparent。

    接下来，将 Blood 纹理拖放到 Texture 框中。完成之后，Inspector 应该如图 8-5 所示。

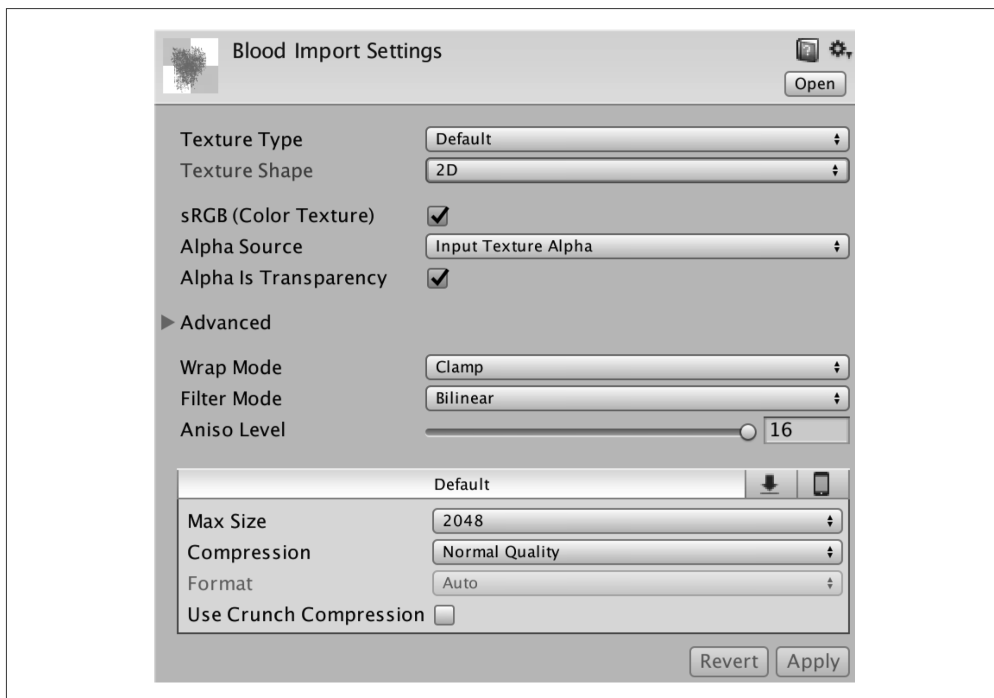


图 8-4：Blood 纹理的导入设置

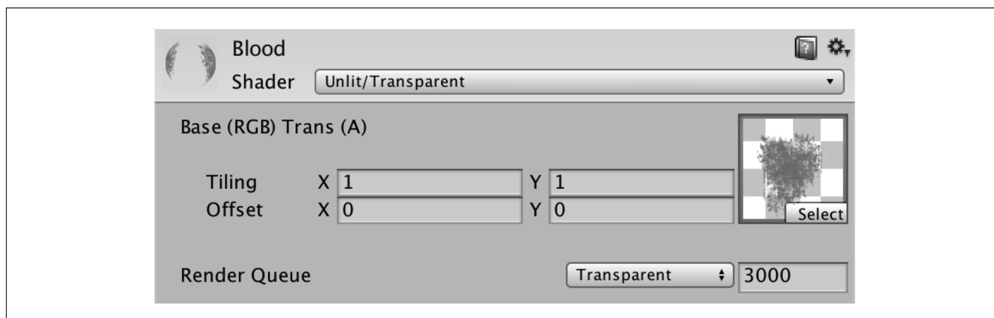


图 8-5：粒子效果的材质

8.2.2 Blood Fountain

现在已经准备好了材质，是时候构建粒子效果了。我们首先构建 Blood Fountain（血流）效果，这将创建一股朝着特定方向发射，并逐渐消失的粒子流。设置步骤如下。

(1) **为粒子系统创建游戏对象**。打开 GameObject 菜单，再打开 Effects 子菜单，然后创建一个新的 Particle System，命名为 Blood Fountain。

(2) **配置粒子系统**。选择对象，并按照图 8-6 和图 8-7 中的设置，更新 Particle System 中的值。



图 8-6：Blood Fountain 的设置

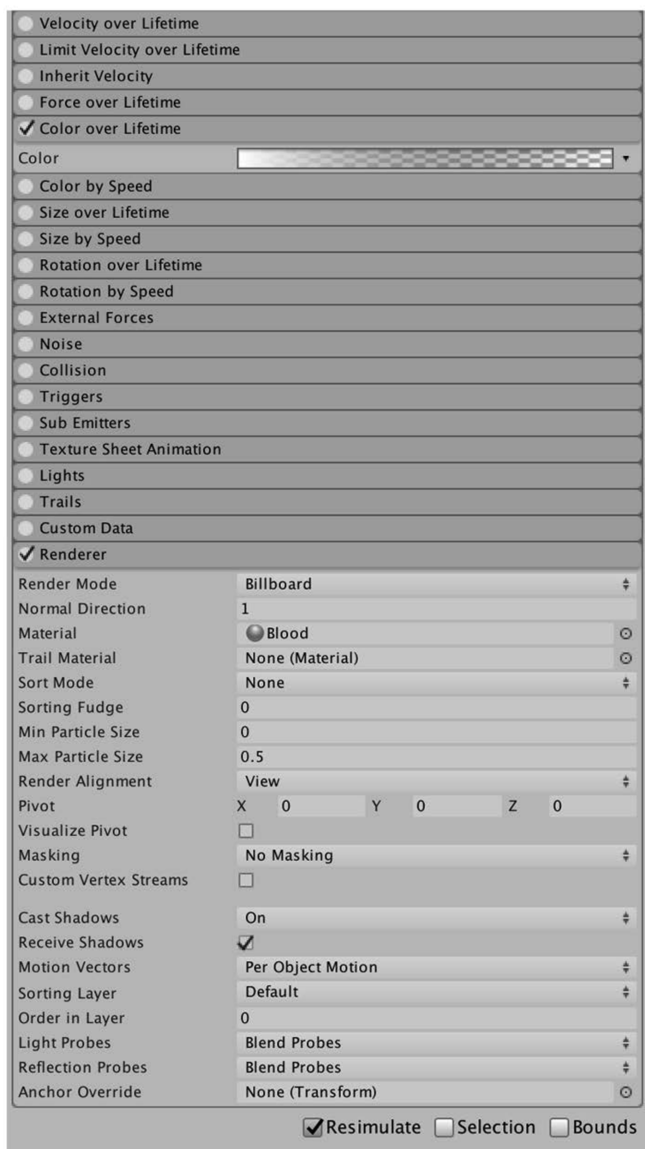


图 8-7：Blood Fountain 的设置（续）

有些参数的值不能直接从屏幕截图中复制，所以需要对它们做一些解释：

- Color over Lifetime 值从开始的 100% Alpha 变为结束时的 0% Alpha，颜色值从开始的白色变为结束时的黑色；
- Particle System 的 Renderer 部分使用刚才创建的 Blood 材质。

(3) **将 Blood Fountain 转换为预设**。将 Blood Fountain 对象拖放到地精文件夹。

8.2.3 Blood Explosion

接下来，我们将创建 Blood Explosion（血爆）预设。它会发射一簇爆炸的粒子，而不是创建一个连续的粒子流。

(1) **创建粒子系统对象**。创建另外一个 Particle System 游戏对象，命名为 Blood Explosion。

(2) **配置粒子系统**。按照图 8-8 的设置，更新 Inspector 中的值。

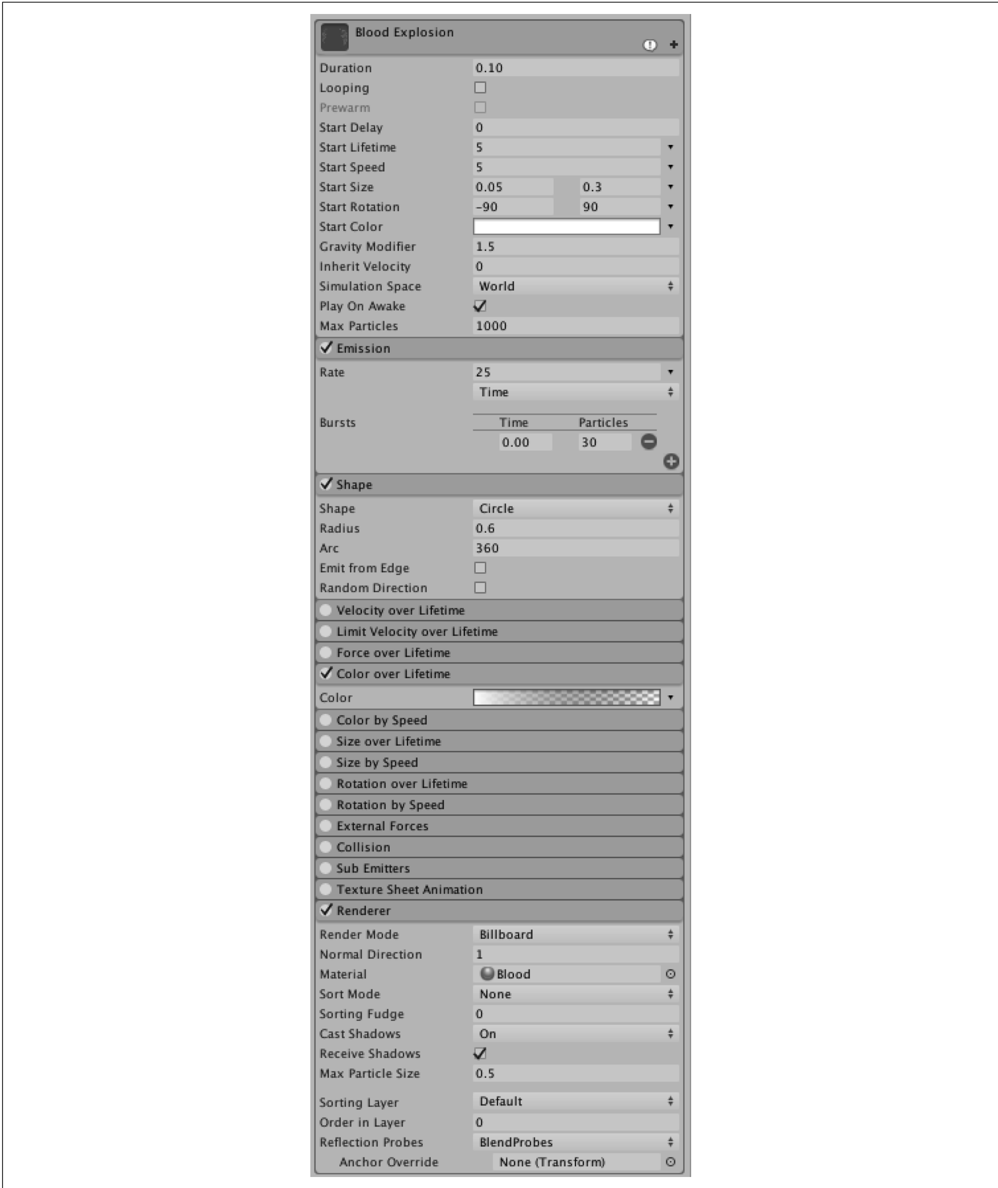


图 8-8：Blood Explosion 的设置

此处使用的材质和 Color over Lifetime 设置与 Blood Fountain 效果相同。唯一区别较大的地方是，此处使用一个圆形发射器，发射率设为一次性发射全部粒子。

(3) **添加 `RemoveAfterDelay` 脚本**。为了保持场景整洁，Blood Explosion 应该在一定时间后移除自身。

为对象添加一个 `RemoveAfterDelay` 组件，并将 Delay 属性设为 2。

(4) **将 Blood Explosion 转换为预设**。

现在基本上就能够在游戏中使用这个粒子系统了。

8.2.4 使用粒子系统

为了让游戏使用这些粒子系统，需要把它们连接到 Gnome 预设，设置步骤如下。

(1) **选择 Gnome 预设**。记得选择正确的预设：使用新的 Gnome 预设，而不是原来的 Prototype Gnome 预设。

(2) **将粒子系统连接到 Gnome**。将 Blood Explosion 预设拖动到 Death Prefab 框，将 Blood Fountain 预设拖动到 Blood Fountain 框。

(3) **测试游戏**。让地精触碰一个陷阱，地精将会流血。

8.3 主菜单

现在，游戏的核心部分已经完成并优化。接下来将构建一些所有游戏都需要的功能，而不是 *Gnome's Well* 特有的功能。也就是说，我们需要一个标题画面，以及从标题画面进入游戏的方法。

这部分将作为一个单独的场景实现，以保持游戏的独立性。因为菜单是比完整的游戏更简单的场景，所以菜单的加载会比整个游戏更快，让玩家能够更快地看到一些东西。另外，菜单将在后台开始加载游戏。当玩家触摸 New Game 按钮时，游戏将完成加载并切换场景。二者结合起来产生的效果是，游戏看起来启动更快。设置步骤如下。

(1) **创建一个新场景**。打开 File 菜单，并选择 New Scene。然后，再次打开 File 菜单，并选择 Save Scene，立即保存新场景。将新场景命名为 Menu。

(2) **添加背景图片**。打开 GameObject 菜单，选择 UI → Image。

将图片的 Source Image 设为 Main Menu Background 精灵。

将图片的锚点设为垂直拉伸，并水平居中。将 x 位置设为 0，Top 边距设为 0，Bottom 边距设为 0，宽度设为 800。

打开图片的 Preserve Aspect，以防止其拉伸。

Inspector 现在看起来应该如图 8-9 所示，背景图片应该如图 8-10 所示。

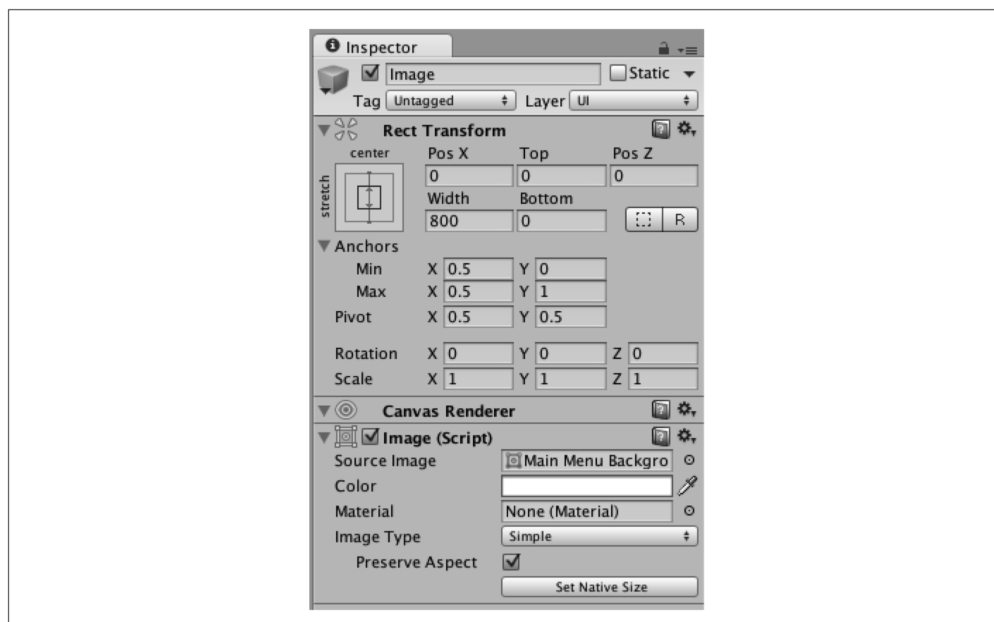


图 8-9：主菜单背景图片的 Inspector



图 8-10：背景图片

(3) **添加 New Game 按钮**。打开 GameObject 菜单，选择 UI → Button。将此对象命名为 New Game。

将按钮的锚点设为 bottom-center。接下来，将 *x* 位置设为 0，*y* 位置设为 40，宽度设为 160，高度设为 30。

将按钮 Lable 的文本设为 New Game。完成之后，按钮应该如图 8-11 所示。



图 8-11：添加按钮后的菜单

加载场景

当玩家触摸 New Game 按钮时，我们想显示一个叠加画面，告诉玩家游戏正在加载。执行下面的步骤进行设置。

(1) **创建叠加画面对象**。创建一个新的空游戏对象，命名为 Loading Overlay，设为 Canvas 对象的子对象。

使叠加画面的锚点垂直和水平拉伸，并将 Top、Bottom、Left 和 Right 边距设为 0。这将使其填充整个屏幕。

(2) **添加一个 Image 组件**。在仍然选中 Loading Overlay 对象的情况下，单击 Add Component 按钮，为其添加一个 Image 组件。画面将填充为白色。

将 Color 属性改为黑色，并设置透明度为 50%。现在叠加画面将成为半透明的黑色画面。

(3) **添加一个标签**。添加一个 Text 对象，设为 Loading Overlay 的子对象。

将标签的锚点设为水平和垂直居中。将 Left、Top、Right 和 Bottom 位置设为 0。

接下来，增加 Text 组件的字号，并使文字垂直和水平居中。将颜色设为白色，将文字设为“Loading...”。

设置好叠加画面后，接下来添加代码。这些代码负责实际加载整个游戏，以及在玩家触摸 New Game 按钮时切换场景。为方便起见，我们将把这段代码添加到 Main Camera 中，不过如果你愿意，也可以添加到一个新的空游戏对象上。设置步骤如下。

在 Main Camera 中添加 `MainMenu` 代码。选择 Main Menu，添加一个新的 C# 脚本，命名为 `MainMenu`。

在 MainMenu.cs 中添加下面的代码：

```
using UnityEngine.SceneManagement;

//管理主菜单
public class MainMenu : MonoBehaviour {

  //包含游戏自身的画面的名称
  public string sceneToLoad;

  //包含“Loading...”文本的UI组件
  public RectTransform loadingOverlay;

  //代表后台加载场景的操作，用于控制场景何时切换
  AsyncOperation sceneLoadingOperation;

  //启动时，开始加载游戏
  public void Start() {

    //确保loading叠加画面不可见
    loadingOverlay.gameObject.SetActive(false);

    //开始在后台加载场景……
    sceneLoadingOperation =
      SceneManager.LoadSceneAsync(sceneToLoad);

    //……但是在准备好之前，还不会实际切换到新场景
    sceneLoadingOperation.allowSceneActivation = false;

  }

  //当触摸New Game按钮时调用
  public void LoadScene() {

    //使loading叠加画面可见
    loadingOverlay.gameObject.SetActive(true);

    //告诉场景加载操作，在完成加载后切换场景
    sceneLoadingOperation.allowSceneActivation = true;

  }

}
```

Main Menu 的脚本负责两个工作：在后台加载游戏场景，以及响应玩家触摸 New Game 按钮的操作。在 `Start` 方法中，让 `SceneManager` 在后台开始加载场景。结果作为一个 `AsyncOperation` 对象（即 `sceneLoadingOperation`）返回，我们可以使用此对象控制如何加载。这里我们告诉 `sceneLoadingOperation`，加载完成时不要激活新场景。这么做意味着，当加载完成时，加载操作将会等待，直到用户准备好进入下一个菜单。

切换到下一个菜单的操作是在 `LoadScene` 方法中完成的，当用户触摸 New Game 按钮时就

会调用此方法。首先，将会显示刚才设置的 loading（加载）叠加画面。然后，告诉场景加载操作，在加载完成后可以激活场景。这么做意味着，如果场景已经完成加载，它会立即显示；如果场景还没有完成加载，它会在加载完成后立即显示。



以这种方式设置主菜单，意味着整个游戏看起来加载得更快。因为主菜单需要加载的资源比主体游戏更少，所以会更快显示出来。当主菜单显示时，用户还要花些时间来点击 New Game 按钮。游戏将在此时加载新场景。不过，因为用户不必盯着一个“请等待”画面，所以相比直接启动游戏而言，这种方式感觉加载得更快。

执行以下步骤。

(1) **配置 Main Menu 组件**。将 Scene to Load 变量设为 Main（即游戏主场景的名称）。将 Loading Overlay 变量设为刚才创建的 Loading Overlay。
(2) **使按钮加载场景**。选择 New Game 按钮，使其运行 Main Camera 的 `MainMenu.LoadScene` 函数。

最后，需要建立构建的场景列表。`Application.LoadLevel` 及其相关的函数只能加载构建的场景列表中包含的场景，也就是说需要确保其中包含 Main 和 Menu 场景。设置步骤如下。

(1) **打开 Build Settings 窗口**。打开 File 菜单，然后选择 File → Build Settings。
(2) **将 Main 和 Menu 场景添加到 Scenes In Build 列表中**。将 Main 和 Menu 场景文件从 Assets 文件夹拖放到 Scenes In Build 列表中。确保 Menu 是列表中的第一项，因为这是游戏启动时应该显示的场景。
(3) **测试游戏**。运行游戏，单击 New Game 按钮，你将进入游戏。

8.4　音效

最后还需要给游戏添加音效。如果没有声音，游戏就只是恐怖的地精死亡画面，所以我们需要解决这个问题。

幸好，我们已经添加到游戏中的代码能够让此事变得简单。当地精触碰到碰撞器时，Signal On Touch 的脚本会播放对应的音效，当然前提是已经关联了音频源。为了实现这一点，需要为各个预设添加 Audio Source 组件。

另外，Game Manager 脚本在地精死亡时，以及地精抱着宝藏成功到达出口时，都会播放音效，因此也需要为 Game Manager 添加 Audio Source 组件。为此，执行下面的步骤。

(1) **为尖刺添加 Audio Source 组件**。找到 SpikesBrown 预设，添加一个新的 Audio Source 组件。
将 Death By Static Object 音效添加到新的 Audio Source。确保关闭 Loop 和 Play On Awake 选项。
为 SpikesRed 和 SpikesBlue 预设重复相同的步骤。
(2) **为 Spinner 添加一个 Audio Source 组件**。找到 Spinner 预设，添加一个新的 Audio Source 组件。将 Death by Moving Object 音效添加到 Audio Source。同样，确保关闭 Loop 和 Play On Awake 选项。

(3) **为 Treasure 添加一个 Audio Source 组件**。找到井底的 Treasure，为其添加一个新的 Audio Source 组件。将 Treasure Collected 音效添加到 Audio Source。同样，确保关闭 Loop 和 Play On Awake 选项。

(4) **为 Game Manager 添加一个 Audio Source 组件**。最后，选择 Game Manager 对象，为其添加一个 Audio Source 组件。保留 Audio Clip 属性为空，将 Game Over 音效添加到 Gnome Died Sound 框中，将 You Win 音效添加到 Game Over Sound 框中。

(5) **测试游戏**。现在，当地精死亡时，拾起宝藏时，以及游戏获胜时，你将听到音效。

8.5 完成游戏后的挑战

现在已经完成了 *Gnome's Well That Ends Well* 的构建，看到的游戏应该类似于图 8-12。祝贺你！

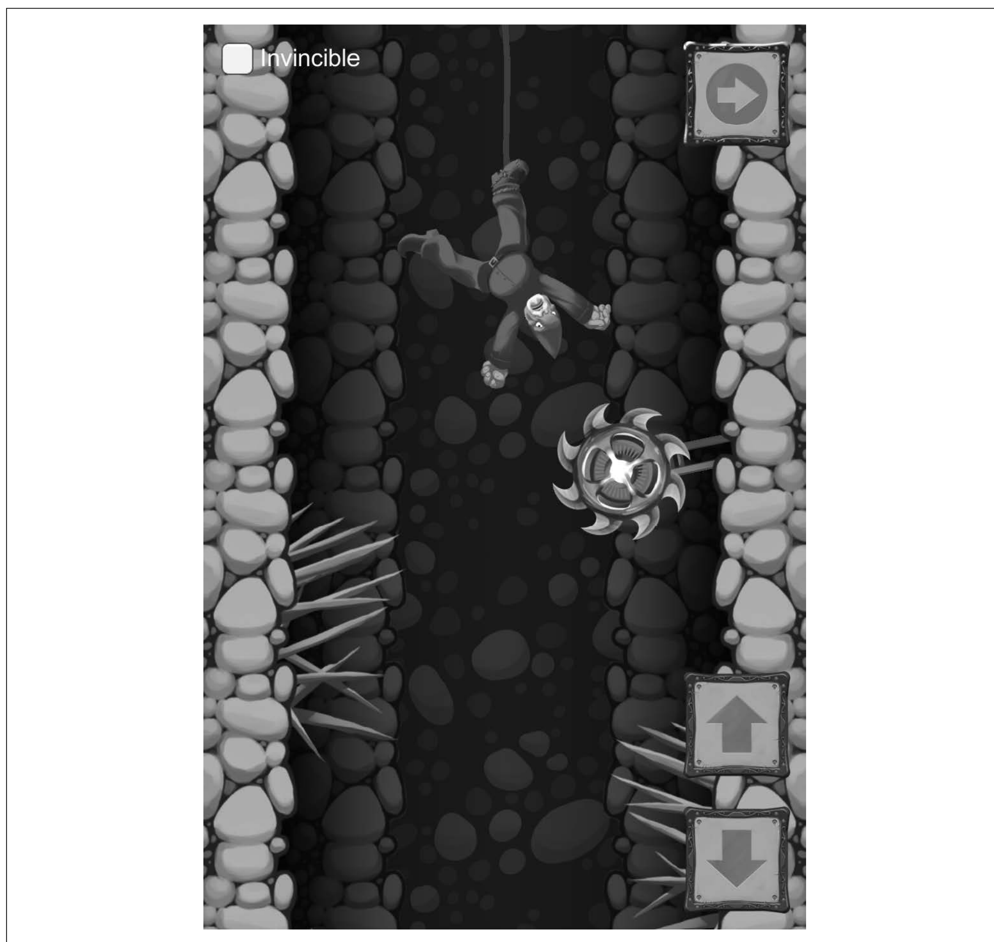


图 8-12：最终的游戏

在此基础上，还可以做一些其他工作来探索这个游戏的更多可能性。

添加鬼魂

在之前的 5.2 节中，我们做了这样的设置：当地精死亡时，会创建一个对象。下一步你可以创建一个预设，显示一个向上漂浮的鬼魂精灵（资源中已包含）。还可以考虑使用粒子效果，使其留下一个淡淡的轨迹。

添加更多陷阱

我们提供了额外两个陷阱的资源：**大镰刀**和**投火器**。大镰刀是一个巨大的刀具，连接到一条锁链上，左右摆动。你需要使用一个 Animator 使其移动。投火器是一个对象，向地精发射火球；当火球击中地精时，应该调用 Game Manager 的 `FireTrapTouched` 函数。别忘了，还可以给地精使用烧焦的骷髅精灵！

构建更多关卡

这个游戏被设计为只有一个关卡，但是可以尝试添加更多关卡。

添加更多效果

使宝藏周围显示粒子（使用 Shiny1 和 Shiny2 图片）。当玩家碰到井壁时，使井壁上掉落粒子。

第三部分

构建一个3D游戏：太空射击游戏

在这一部分中，我们将从头构建另外一个游戏。与第二部分构建的游戏不同，这个游戏是一个 3D 游戏。我们将构建一个太空大战模拟器，玩家需要保卫一个空间站，使其避免被飞来的小行星击毁。在构建游戏的过程中，我们将探索其他游戏中经常出现的系统，例如射击炮弹、重生对象及管理 3D 模型的外观。这个游戏会很棒！

第9章 构建一个太空射击游戏

Unity 不仅是构建 2D 游戏的出色平台，对于构建 3D 内容而言也极其优秀。Unity 最初被设计为一个 3D 引擎，后来才添加 2D 功能，所以 Unity 的功能一开始就是为 3D 游戏构建的。

本章中，我们将学习如何使用 Unity 构建 *Rockfall*——一个 3D 太空模拟游戏。这种类型的游戏在 20 世纪 90 年代中期很受欢迎，当时《星球大战：X-Wing》（1993）和《天旋地转：无限空间》（1998）这两款游戏让玩家能够在太空中自由飞翔、射击坏人、四处轰炸。这类游戏与飞行模拟器有着密切的关系，不过由于玩家并不期望它们完全符合真实飞行中的物理定律，使得游戏开发人员能够采用更加偏重趣味性的机制。



并不是说不存在街机风格的飞行模拟游戏，只不过街机风格的空战模拟游戏要比接近真实物理定律的模拟游戏更为常见。《坎巴拉太空计划》是近年来最突出的例外，其空间飞行物理模拟极其真实，因此与本章要介绍的游戏类型相去甚远。如果你确实想学习轨道力学，以及在远重心点施加顺行推力时会发生什么，那么那种游戏很适合你。

因此，虽然本章要介绍的游戏类型常常被称为“太空大战模拟游戏”，但是称其为“太空模拟游戏”其实是相当合理的。

聊了不少关于名称的问题，下面我们就来发射激光束。

结束接下来的几章的内容后，我们将得到如图 9-1 所示的一个游戏。

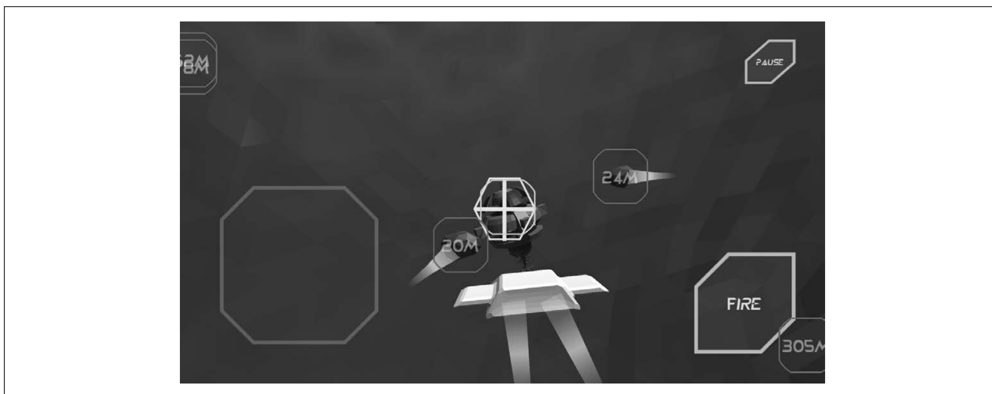


图 9-1：完成后的游戏

9.1 设计游戏

开始设计游戏时，我们决定了如下的一些关键约束。

- 每局游戏时长限制在几分钟以内。
- 控制机制应该非常简单，尽量只保留最基本的“移动”和“发射”功能。
- 游戏应该关注多个短期挑战，而不是单个大挑战。也就是说，有许多小敌人，而不是进行一次 boss 战（这与第二部分讨论的 2D 游戏 *Gnome's Well* 正相反）。
- 游戏玩法主要是在太空中发射激光束。在太空中发射激光束的电子游戏永远都不嫌多。

借助纸笔来思索高层次概念几乎每次都行得通。这是一种非结构化的方法，有助于发现适合总体计划的新点子。于是我们坐下来，把对游戏的基本构想迅速绘制了出来，如图 9-2 所示。

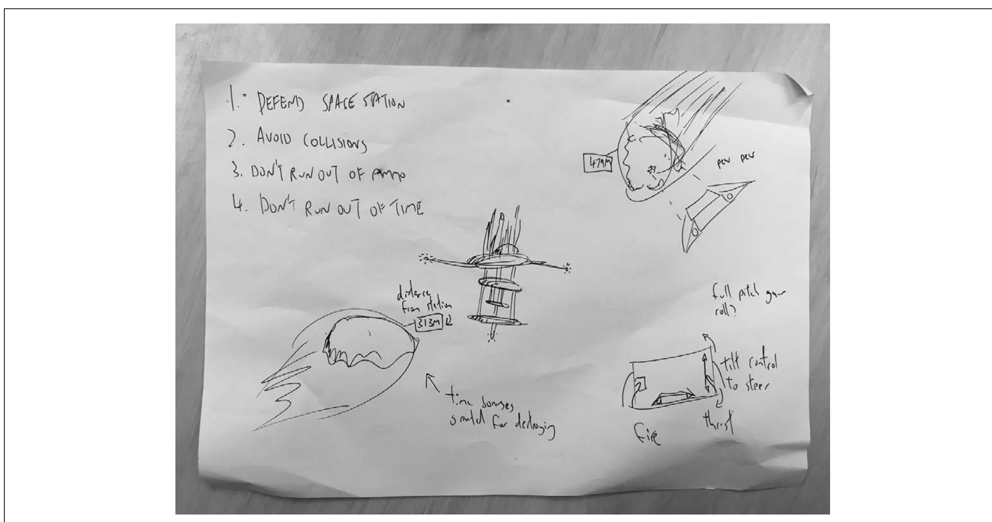


图 9-2：绘制出对游戏最初的构想

我们很快完成了这个素描图，并有意使其很粗略，但是从中可以看出额外的几点：小行星朝向一个空间站移动，玩家使用屏幕上的摇杆操控飞船，并按开火按钮发射激光束。还可以看到一些更加具体的细节，这是思考如何表现这类场景的结果，例如用一个标签显示小行星距离空间站多远，以及考虑玩家如何拿着设备。

有了这个草图后，我们找到了 Rex Smeal，一位艺术家朋友，让他把这个杂乱的草图改为更加丰满的绘图。尽管这并不是游戏设计过程中的紧要环节，但帮助我们确定了游戏的整体感觉。我们尤其意识到，需要重点关注玩家要防御的空间站，因为空间站应该看起来像一个值得保护的地方。在找到这个艺术家朋友并对他描述了游戏后，他为我们做了如图 9-3 所示的设计。当我们共同敲定设计后，Rex 对设计进行了一些改善，使其能够被建模（如图 9-4 所示）。

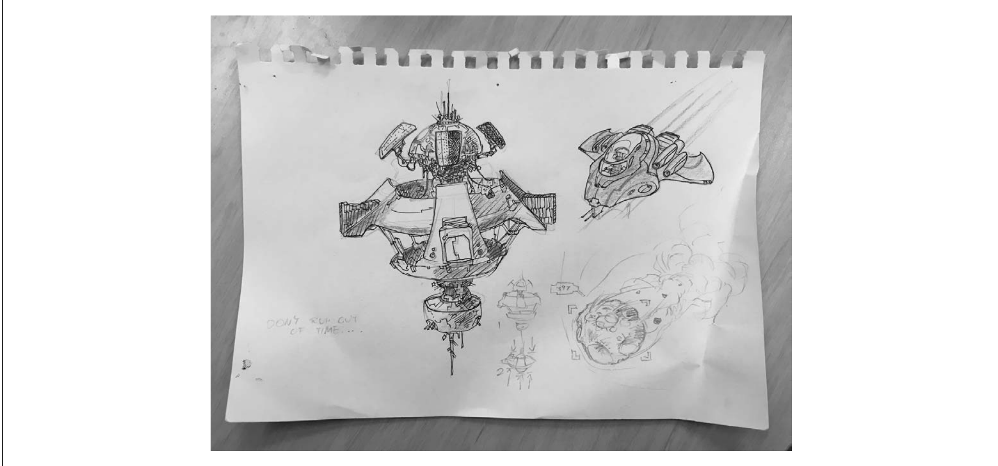


图 9-3：Rex 最初设计的游戏外观概念图

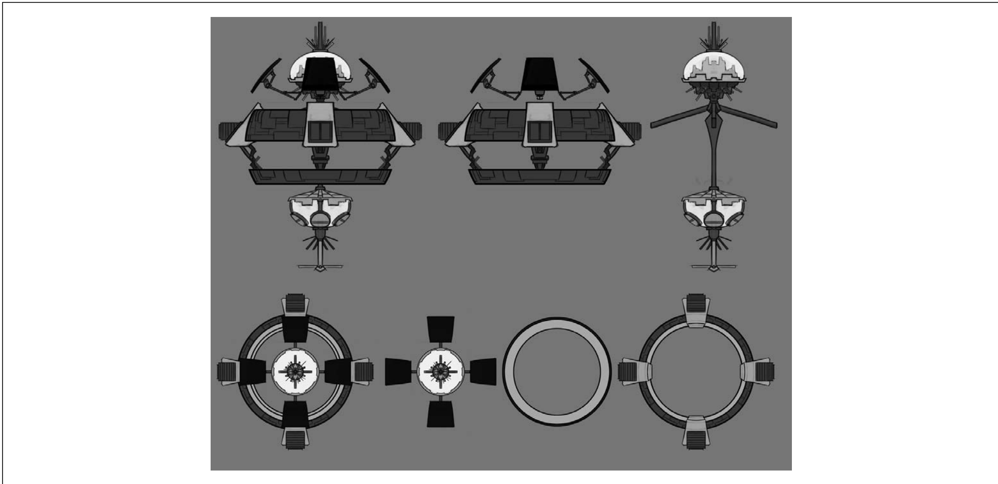


图 9-4：改善后的空间站概念图，能够被建模

我们采用了这种设计，并在 Blender 中进行建模。在开发空间站时，考虑到其简约的风格，我们认为使用低多边形的方法就很好（这一点受到了 Heather Penn 和 Timothy Reynolds 等艺术家的启发）。（并不是说低多边形作品很简单或很容易制作，只是说这种风格用起来更顺手，就像使用铅笔绘图比使用油彩更轻松一样。）

图 9-5 显示了这个空间站的样子。另外，我们还在 Blender 中建模了飞船和一个小行星，分别如图 9-6 和图 9-7 所示。

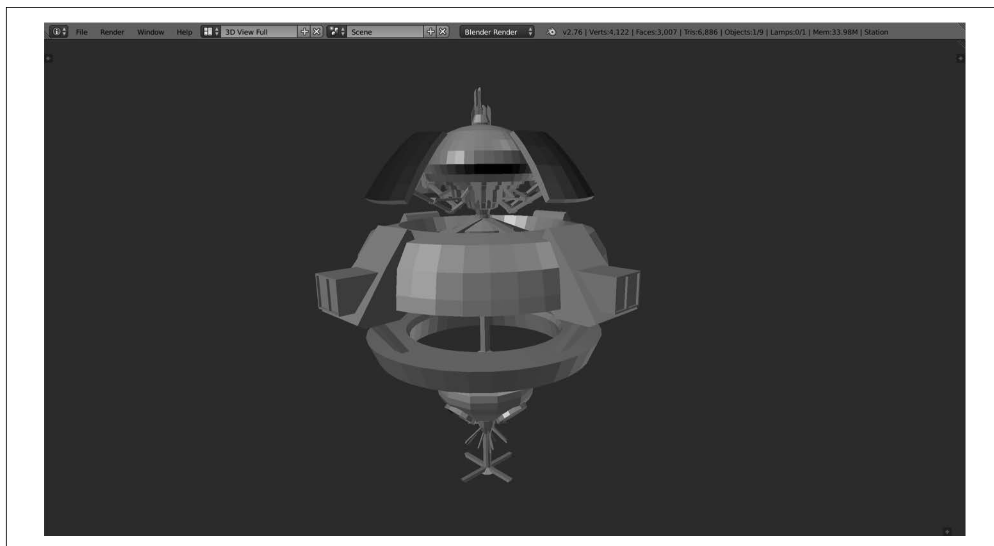


图 9-5：建模后的空间站

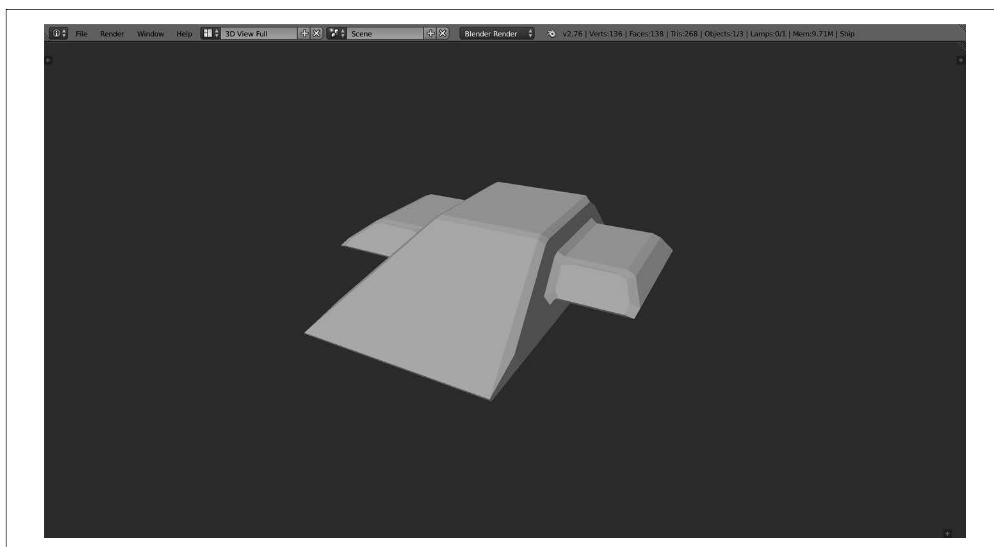


图 9-6：建模后的飞船

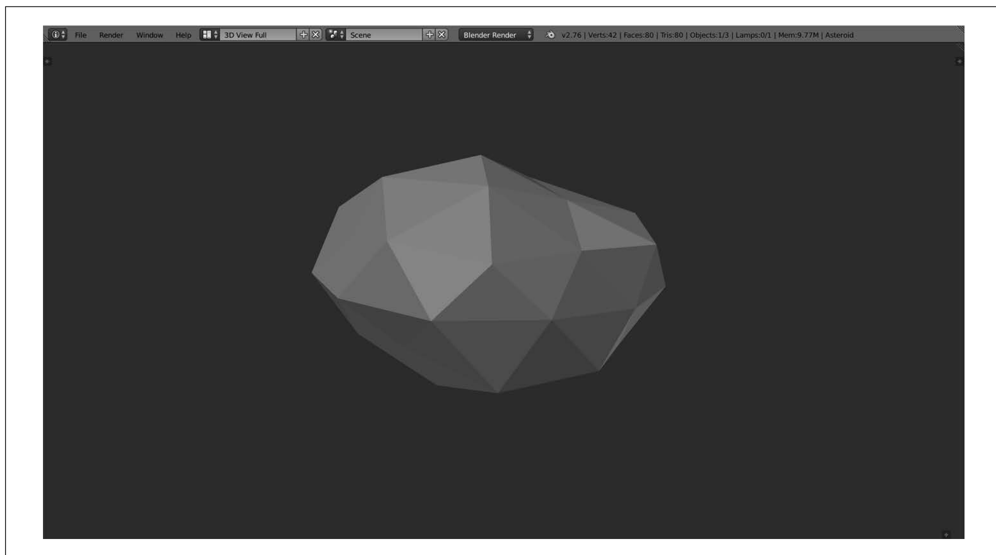


图 9-7：建模后的小行星

获取资源

在构建这个游戏的过程中，将用到多种资源，包括音效、模型以及纹理，我们已经为你打包好了这些资源。首先，你需要下载这些资源。我们将其整理存放在文件夹中，方便你找到需要的资源。

从本项目的 GitHub 页面（https://raw.githubusercontent.com/thesecretlab/MobileGameDevWithUnity1stEd/master/3D-Game/Packages/3D-Game.unitypackage）可下载这些资源。[1]

9.2　架构

游戏架构的核心与 *Gnome's Well That Ends Well* 非常类似。有一个游戏管理器负责实例化关键的游戏对象，例如玩家控制的飞船和空间站；当玩家死亡时，游戏会结束，此时游戏管理器也会得到通知。

这个游戏中的用户界面比我们之前创建的稍微复杂一些。在 *Gnome's Well* 中，游戏内的控制方式只有上下两个按钮，再加上倾斜控制；而在 3D 游戏中，玩家可以在任意方向上移动，此时倾斜控制的效果不是太好。因此，本游戏将使用一个屏幕“摇杆”，这是屏幕上的一个区域，负责检测触摸，并允许用户拖动手指来指示方向。这些信息将被提供给一个共享的输入管理器，飞船使用此输入管理器来调整自己的飞行方向。

注 1：也可从本书图灵社区页面（http://www.ituring.com.cn/book/2117）的“随书下载”处点击下载。
——编者注



在 3D 游戏中实现倾斜控制更具挑战性，但并不意味着不能做好。*NOVA 3* 是一个 FPS（第一人称射击类）游戏，使用倾斜控制来让玩家改变角色的朝向并精确地瞄准目标。可以玩玩这个游戏，来了解一下他们如何获取输入。

我们故意让游戏中使用的飞行模型不完全符合现实。最简单、最符合现实的方法是建模一个物理对象，使其应用一个前向推力，并使用物理力来旋转飞船。但是，操控这种飞船十分困难，玩家很容易就会迷失方向。因而，我们决定实现伪物理：飞船将始终以固定的速度向前移动，并且没有动量。另外，玩家不能翻滚飞船，任何翻滚将被纠正（也就是说，不同于真实的外太空，这个游戏中的太空有一个"向上方向"）。



设计与指导

我们为本书中的游戏做出的每个决定都是完全主观的。虽然我们决定不为这个游戏实现一个物理飞行模型，但这并不意味着在街机风格的飞行模拟中应该避免物理飞行模型。你可以自由尝试，看看自己会有什么好主意。不要因为某个图书作者的言论，就认为游戏是固定一种模式。他们可能是在撰写图书的过程中设想出了一种模式。

小行星是预设，由专门的"小行星生成器"对象创建。该对象将不时地创建小行星实例，并使它们的方向朝向空间站。当小行星与空间站碰撞时，将降低空间站的生命值；当空间站的生命值为 0 时，就被摧毁，游戏将结束。

9.3　创建场景

首先设置场景。我们将创建一个新的 Unity 项目，然后创建在场景中飞行的飞船。首先执行下面的步骤。

(1) **创建项目**。创建一个新的 Unity 项目，命名为 Rockfall，将其模式设为 3D，如图 9-8 所示。

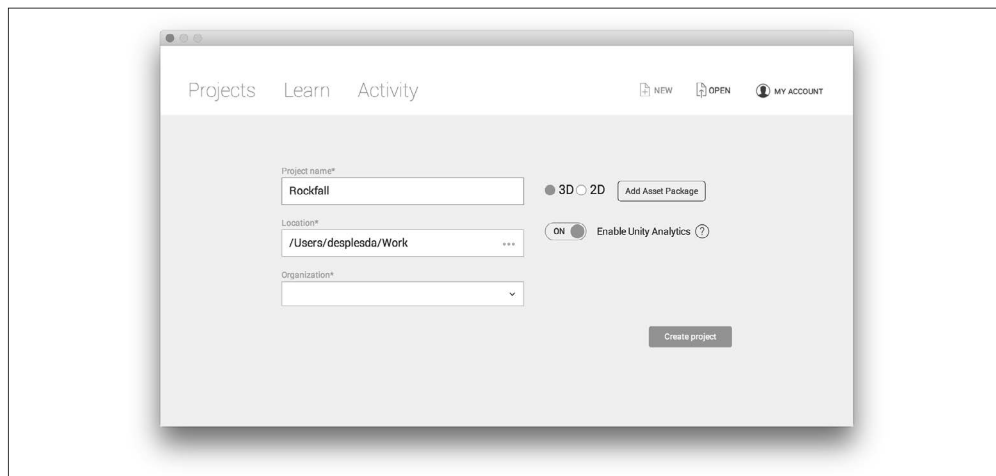


图 9-8：创建项目

(2) **保存新场景**。当 Unity 创建了项目并显示空场景之后，打开 File 菜单并选择 Save，保存新场景。将新场景命名为 Main.scene，保存到 Assets 文件夹中。

(3) **导入下载的资源**。双击 9.1 节“获取资源”中下载的 .unitypackage 文件，将全部资源导入到项目中。

现在就可以开始构建飞船了。

9.3.1 飞船

首先创建飞船，这将用到下载资源中的飞船模型。

Ship 对象本身是不可见的，只包含脚本。它将被附加多个子对象，这些子对象负责处理在屏幕上显示的具体任务。

(1) **创建 Ship 对象**。打开 GameObject 菜单，选择 Create Empty。场景中将出现一个新的 `GameObject`，将其重命名为 Ship。

现在来添加飞船的模型。

(2) **打开 Models 文件夹，将 Ship 模型拖放到 Ship 游戏对象上**。这将把飞船的 3D 模型添加到场景中，如图 9-9 所示。把它拖放到 Ship 游戏对象上以后，它将成为一个子对象，这意味着它将随着 Ship 父游戏对象一起移动。

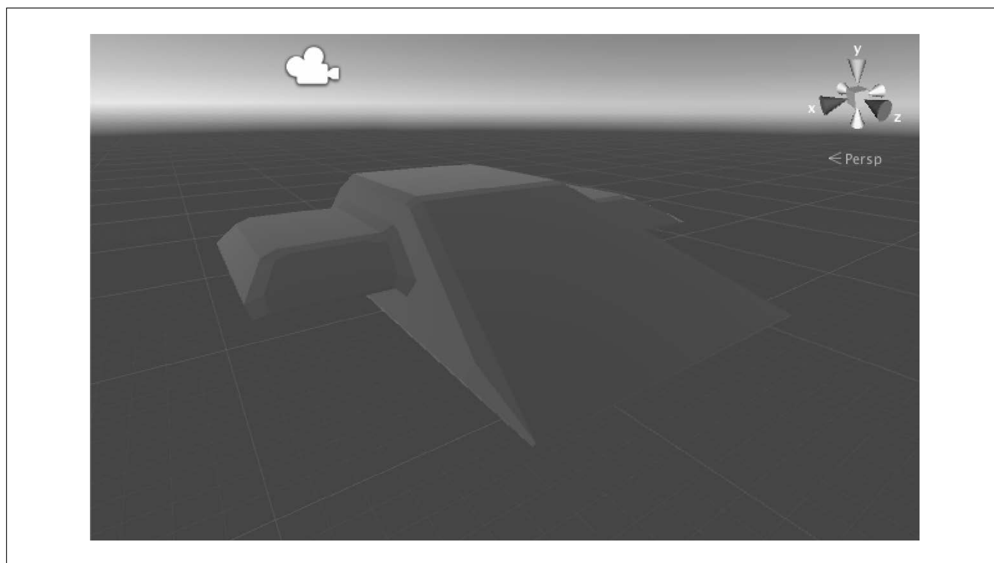


图 9-9：场景中的飞船模型

(3) **将模型对象重命名为 Graphics**。

接下来，我们需要让 Graphics 对象与 Ship 对象位于相同的位置。

(4) **选择 Graphics 对象**。单击 Transform 组件右上角的齿轮图标，选择 Reset Position，如图 9-10 所示。



保留旋转设置为 (−90, 0, 0)。这是因为飞船是在 Blender 中建模的，而 Blender 使用的坐标系统与 Unity 不同。具体来说，Blender 的“上”方向是 *z* 轴，而 Unity 的“上”方向是 *y* 轴。为了解决这个问题，Unity 自动旋转 Blender 模型来进行补偿。

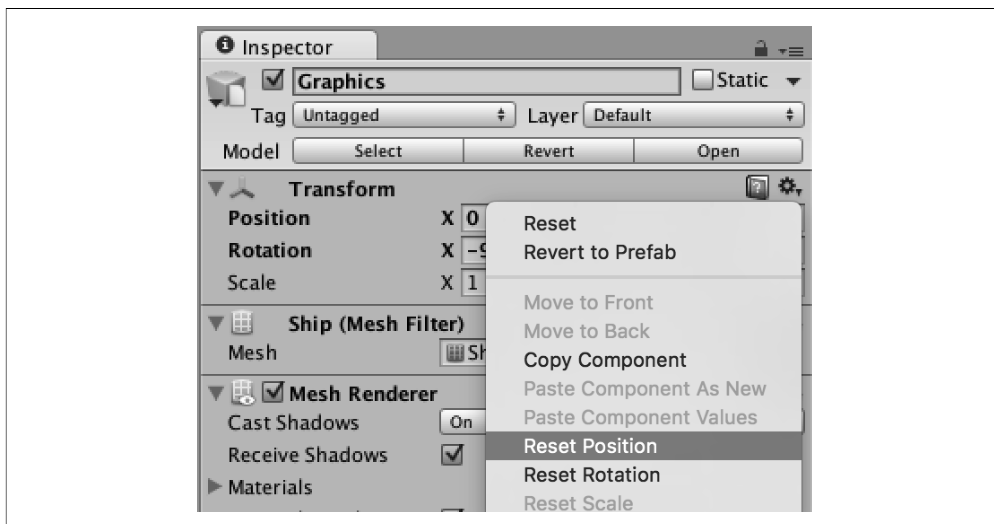


图 9-10：重设 Graphics 对象的位置

我们想让飞船与对象碰撞。为此，将添加一个碰撞器。

(5) **为飞船添加一个 Box Collider 组件**。选择 Ship 对象，即 Graphics 对象的父对象，然后单击 Inspector 底部的 Add Component 按钮。选择 Physics → Box Collider。
添加了碰撞器以后，打开 Is Trigger，并将碰撞盒的 Size 设为 (2, 1.2, 3)，这将创建一个包围玩家的盒子。

飞船需要以恒定速度向前移动。为此，我们将添加一个脚本，用于移动其所附加到的对象。

(6) **添加 `ShipThrust` 脚本**。选择 Ship 对象，单击 Inspector 底部的 Add Component 按钮。创建一个新的 C# 脚本，命名为 ShipThrust.cs。
然后，打开 ShipThrust.cs，添加下面的代码：

```
public class ShipThrust : MonoBehaviour {

  public float speed = 5.0f;
  //使飞船以恒定速度向前移动
  void Update () {
    var offset = Vector3.forward * Time.deltaTime * speed;
    this.transform.Translate(offset);
  }
}
```

ShipThrust 脚本提供了一个参数 speed，Update 函数将使用该参数向前移动对象。此前向移动是将前向向量与 speed 参数和 Time.deltaTime 属性相乘的结果，这将确保对象以相同的速度向前移动，而与每秒调用多少次 Update 函数无关。



确保把 ShipThrust 组件附加到 Ship 对象，而不是 Graphics 对象。

(7) **测试游戏**。单击 Play 按钮，可看到飞船向前移动。

摄像机跟随

下一步是让摄像机跟随飞船移动。有几种可选的方法，最简单的是将摄像机放到 Ship 对象内，这样摄像机就会随着飞船一起移动。但是，这种效果看起来不是太好，因为这样一来，飞船无论什么时候也无法相对于摄像机旋转。

更好的方法是将摄像机作为独立的对象，然后添加一个脚本，使摄像机随着时间慢慢移动到正确的位置。这意味着当飞船急转弯时，摄像机过会儿才能跟上，这也是现实世界中摄像师在跟拍一个移动对象时的效果。

(1) **向主摄像机添加 SmoothFollow 脚本**。选择 Main Camera，然后单击 Add Component 按钮。添加一个新的 C# 脚本，命名为 SmoothFollow.cs。
打开该文件，添加下面的代码：

```
public class SmoothFollow : MonoBehaviour
{

  //我们跟随的目标
  public Transform target;

  //我们想让摄像机位于目标上方的高度
  public float height = 5.0f;

  //与目标的距离，忽略高度
  public float distance = 10.0f;

  //减缓旋转和高度变化的程度
  public float rotationDamping;
  public float heightDamping;

  //每一帧调用一次Update
  void LateUpdate()
  {
    //如果没有目标，就返回
    if (!target)
      return;

    //计算当前的旋转角度
    var wantedRotationAngle = target.eulerAngles.y;
    var wantedHeight = target.position.y + height;
```

```
        //记录我们当前的位置和观看方向
        var currentRotationAngle = transform.eulerAngles.y;
        var currentHeight = transform.position.y;

        //减小y轴附近的旋转
        currentRotationAngle
          = Mathf.LerpAngle(currentRotationAngle,
            wantedRotationAngle,
              rotationDamping * Time.deltaTime);

        //减小高度
        currentHeight = Mathf.Lerp(currentHeight,
          wantedHeight, heightDamping * Time.deltaTime);

        //将角度转换为旋转
        var currentRotation
          = Quaternion.Euler(0, currentRotationAngle, 0);

        //将摄像机在x-z平面上的位置设为目标后方distance米处
        transform.position = target.position;
        transform.position -=
          currentRotation * Vector3.forward * distance;

        //使用新的高度设置摄像机的位置
        transform.position = new Vector3(transform.position.x,
          currentHeight, transform.position.z);

        //最后，观看目标面对的方向
        transform.rotation = Quaternion.Lerp(transform.rotation,
          target.rotation,
            rotationDamping * Time.deltaTime);

    }
}
```



本书中使用的 SmoothFollow.cs 脚本基于 Unity 提供的代码。我们对其稍做调整，使其更加适合飞行模拟游戏。如果你想查看这段代码的原始版本，可以在 Unity 的包中找到。通过打开 Assets 菜单并选择 Import Package → Utility，可以导入 Unity 的包。导入后，在 Standard Assets → Utility 中可找到 SmoothFollow.cs 文件的原始版本。

`SmoothFollow` 首先在 3D 空间中计算出摄像机应该在什么位置，然后计算该位置与摄像机当前位置**之间**的一个点。应用到多个帧时，其效果就是让摄像机逐渐接近该点，但在比较靠近时会停下来。另外，因为在每一帧中，摄像机应该在的位置都会改变，所以摄像机总是会稍微滞后一些，而这正是我们想要的效果。

(2) **配置 SmoothFollow 组件**。将 Ship 对象拖放到 Target 字段中。

(3) **测试游戏**。单击 Play 按钮。当游戏启动时，Game 面板不会再显示飞船移动，摄像机将会跟随飞船移动。在 Scene 面板中可以看到这种效果。

9.3.2 空间站

遭受小行星撞击威胁的空间站，将采用与飞船相同的开发模式：创建一个空的游戏对象，并向其附加一个模型。空间站比飞船简单一些，因为它是完全消极的：它只是停留在那里，有飞石不断冲向它。按照下面的步骤进行设置。

(1) **为空间站创建一个容器**。创建一个新的空游戏对象，命名为 Space Station。
(2) **将模型添加为一个子对象**。打开 Models 文件夹，将 Station 模型拖放到 Space Station 游戏对象上。
(3) **重置 Station 模型对象的位置**。选择刚才添加的 Station 对象，右击 Transform 组件。选择 Reset Position，就像对 Ship 模型执行的操作一样。

完成之后，空间站应该如图 9-11 所示。

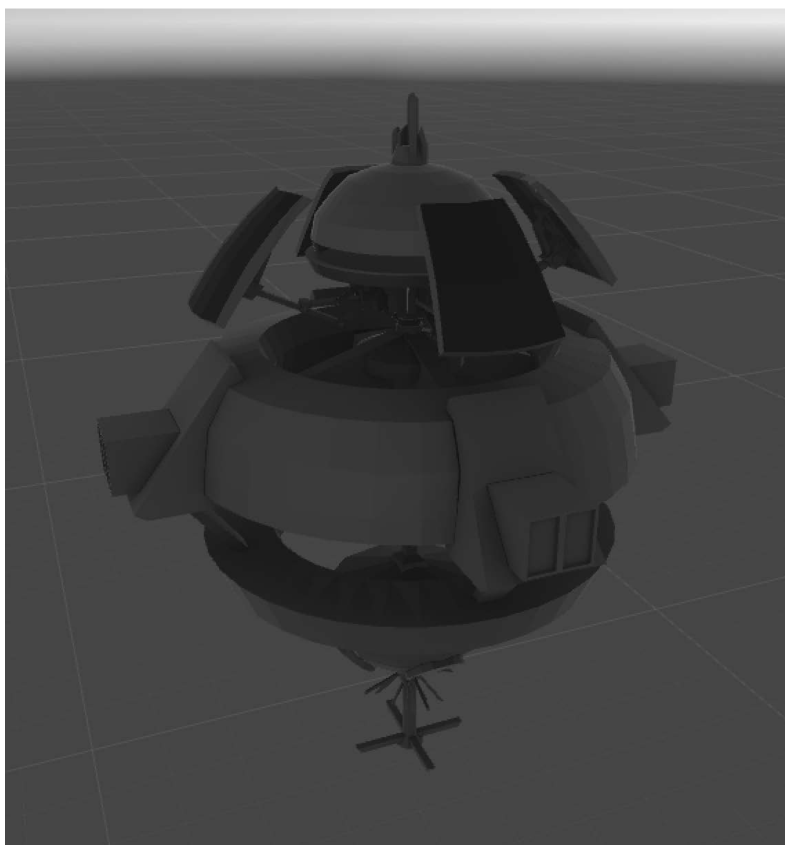


图 9-11：空间站

添加了模型后，我们快速看看模型的结构，并确保模型有碰撞器。空间站的碰撞器很重要，因为小行星（后面会添加）需要有东西与之碰撞。

选择模型对象并展开，以显示其子对象。空间站模型由多个子网格组成，主子网格是 Station。选中 Station。

查看 Inspector。除了 Mesh Filter 和 Mesh Renderer 以外，还可以看到一个 Mesh Collider（如图 9-12 所示）。如果**没有**看到，请查看“模型与碰撞器”附注栏。

图 9-12：空间站的碰撞器



模型与碰撞器

当导入模型时，Unity 也可以为其创建一个碰撞器。从 Asset 包导入模型时，也导入了我们为其创建的设置，包括为空间站创建一个碰撞器的设置（我们也为 Ship 和 Asteroid 模型创建了这种设置）。

如果没有看到，或者如果你在导入自己的模型，想要知道如何设置，则可以选择模型自身（即 Models 文件夹中的文件），并查看和修改设置（如图 9-13 所示）。特别要注意，图中选择了 Generate Colliders 选项。

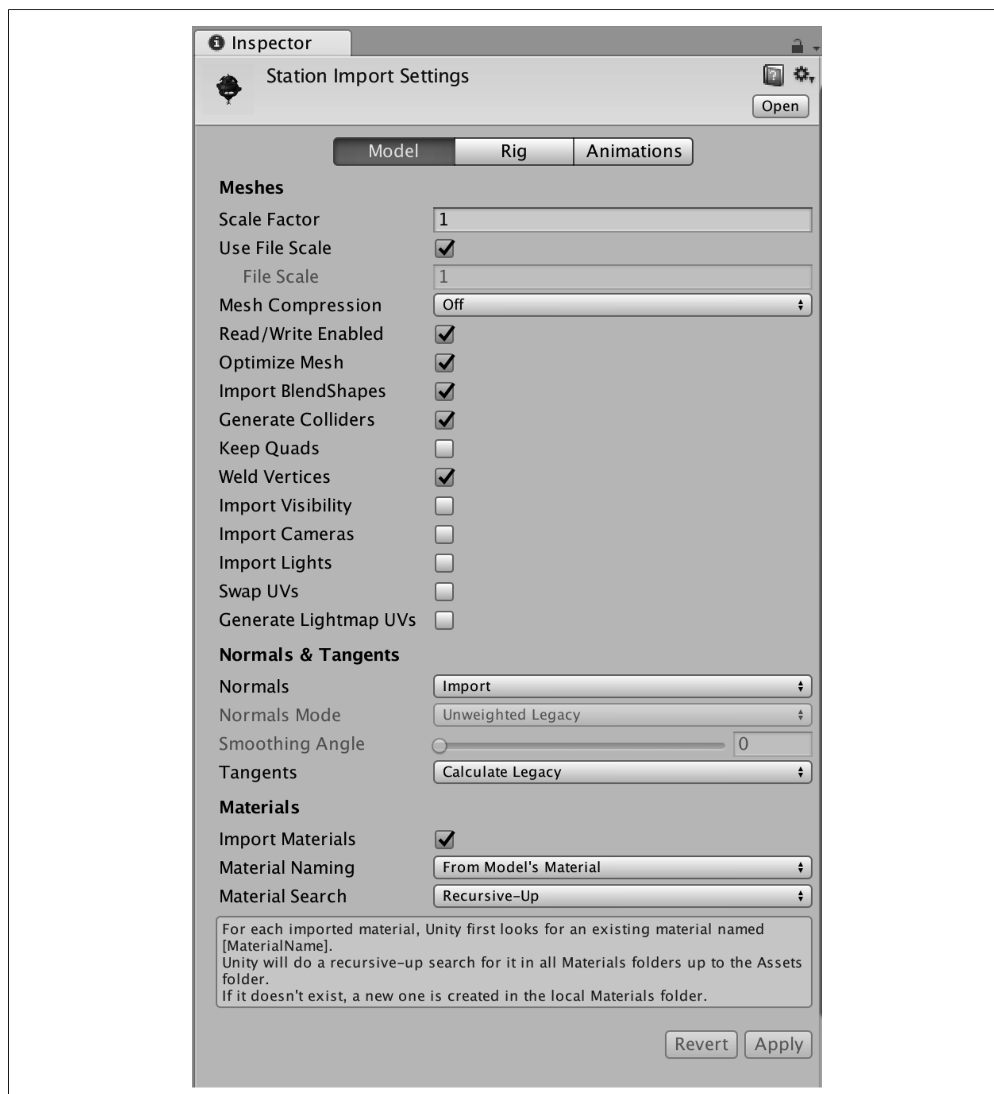


图 9-13：空间站模型的导入设置

9.3.3　天空盒

目前，天空盒（skybox）是 Unity 的默认设置，用于背景在星球表面上的游戏。改变这个天空盒，使玩家看上去像是漂浮在太空中，对于让游戏传递正确的感受很重要。

天空盒会创建一个虚拟立方体，这个虚拟立方体总是在场景中的其他对象之下绘制，并且从不会相对摄像机移动。这样产生的效果就是，立方体上的纹理看上去无限远，所以才叫作天空盒。

为了让玩家看起来像在一个球体中（而不是一个盒状立方体中），需要扭曲天空盒上的纹理，使边缘部分没有可见的接缝。有多种实现方法，包括 Adobe Photoshop 的几个插件，但是这些方法大部分被设计为将你或其他人拍摄的照片扭曲处理。找到为电子游戏设计的太空照片并不容易；相反，使用工具来创建这种图片就简单多了。

创建天空盒

有了天空盒的图片后，就可以把它们添加到游戏中。具体做法是，创建一个天空盒材质，然后把该材质提供给场景的光照设置。需要执行的步骤如下。

(1) **创建 Skybox 材质**。打开 Assets 菜单，选择 Create → Material，创建一个新的材质，将其命名为 Skybox，然后移动到 Skybox 文件夹中。

(2) **配置材质**。选择该材质，将着色器从 Standard 改为 Skybox → 6 Sided。Inspector 将发生改变，允许附加 6 个材质，如图 9-14 所示。

在 Skybox 文件夹中找到天空盒纹理，并将这 6 个天空盒纹理拖放到对应的框中：将 Up 纹理放到 Up 框中，将 Front 纹理放到 Front 框中，以此类推。

完成之后，Inspector 应该如图 9-15 所示。

(3) **将天空盒连接到光照设置**。打开 Window 菜单，选择 Lighting → Settings，Lighting 面板将会显示。在靠近面板顶部的位置，可以看到一个 Skybox 框。将刚才附加的 Skybox 材质放到该框中，如图 9-16 所示。

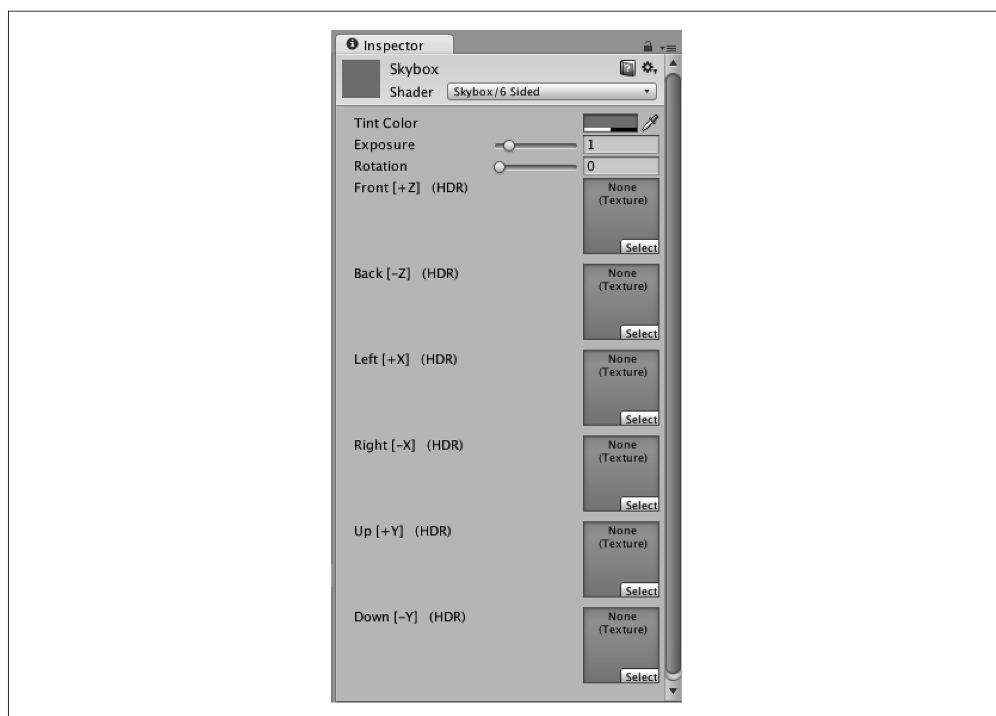


图 9-14：没有纹理的天空盒

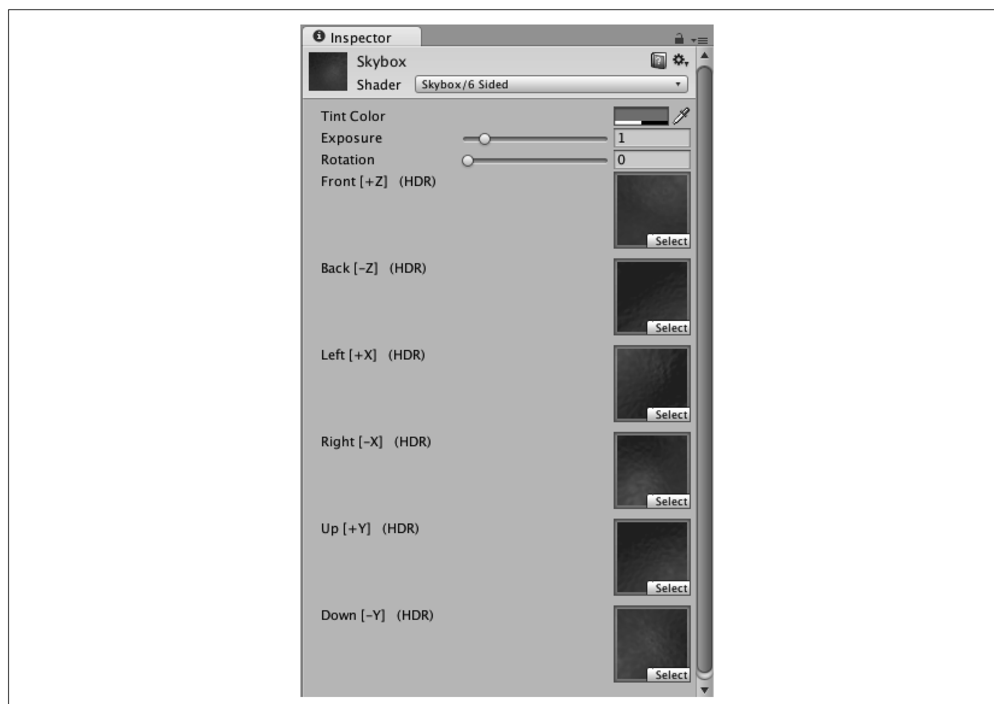


图 9-15：附加纹理后的天空盒

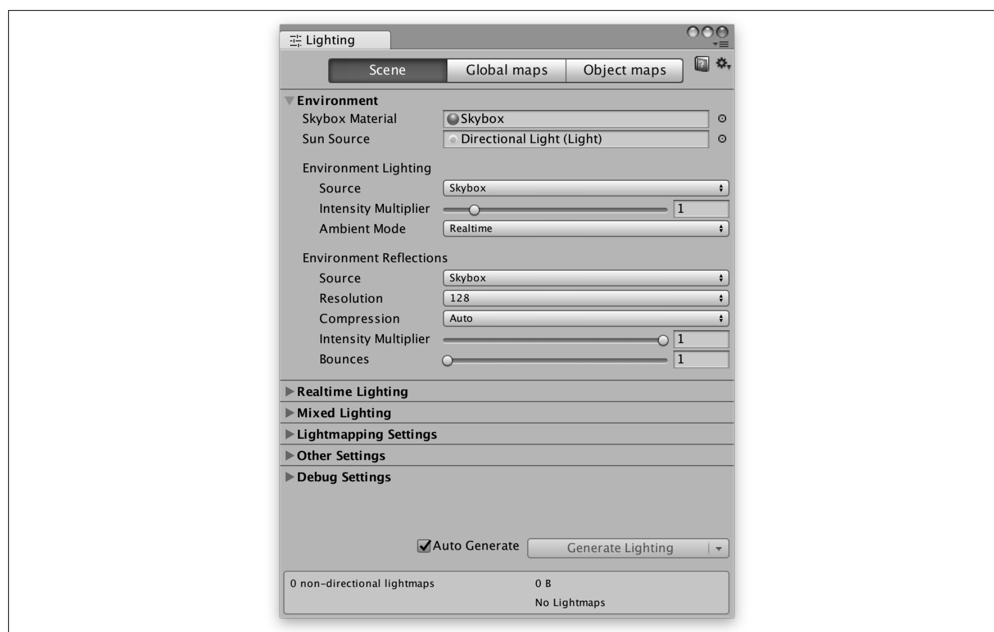


图 9-16：创建光照设置

完成之后，天空将被替换为太空图片，如图 9-17 所示。另外，Unity 的光照系统使用天空盒的信息来影响对象的照明方式。如果仔细观察，会注意到飞船和空间站都带点绿色调，这是因为天空盒图片是绿色的。

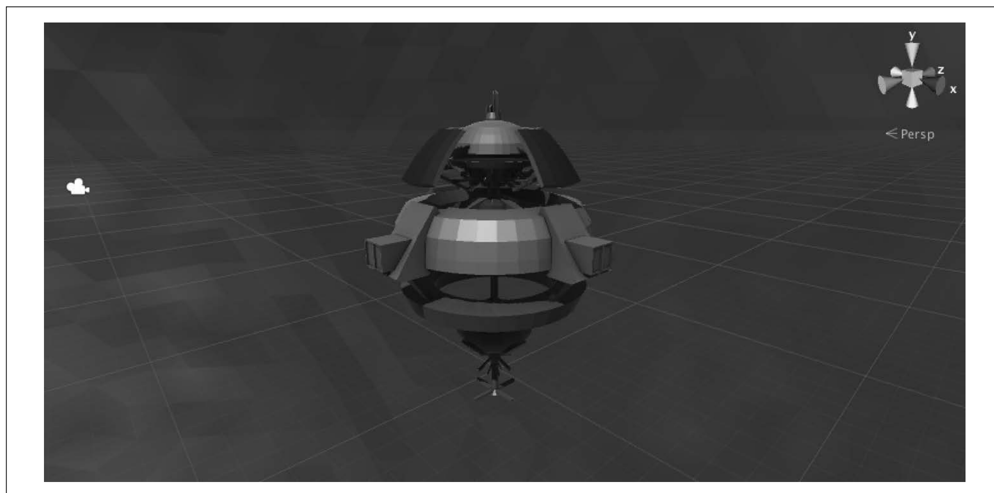


图 9-17：使用了天空盒（另见彩插）

9.3.4 画布

目前，飞船在太空中将始终向前移动，这是因为玩家还没有控制飞行的方法。我们很快将添加一个控制飞船的 UI，但是在那之前，我们需要创建和设置好用来显示 UI 的画布，步骤如下。

(1) **创建画布**。打开 GameObject 菜单，选择 UI → Canvas。这将创建一个 Canvas 和一个 EventSystem 对象。

(2) **配置画布**。选择 Canvas 游戏对象，在附加的 Canvas 组件的 Inspector 中，找到 Render Mode 设置，将其改为 Screen Space-Camera。新的选项将会显示，允许提供相对于此渲染模式的具体设置。

将 Main Camera 拖放到 Render Camera 框中，并将 Plane Distance 改为 1，如图 9-18 所示。这将把 UI 画布放到刚好距离摄像机一个单位的地方。

将 Canvas Scaler 的 UI Scale Mode 设置改为 Scale with Screen Size，并将 Reference Resolution 设为 1024 × 768，这是适合 iPad 的形状大小。

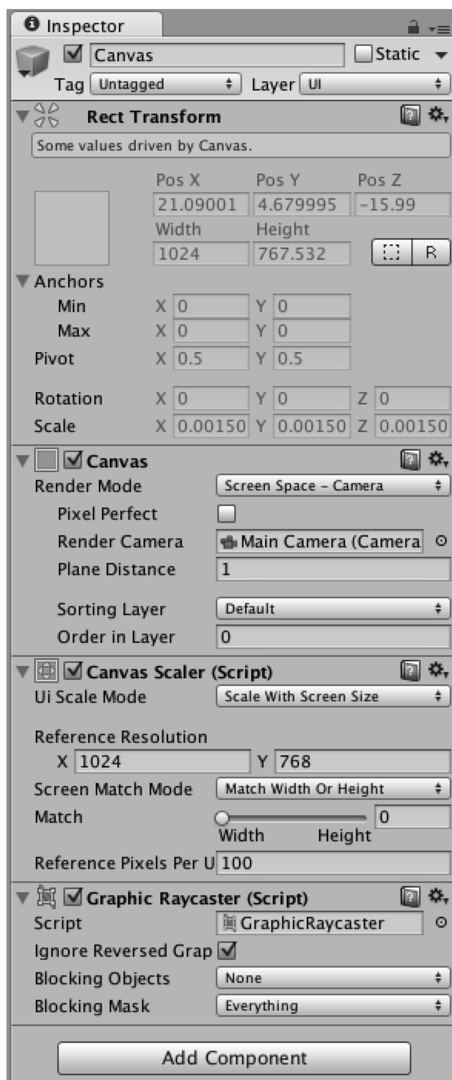


图 9-18：画布的 Inspector

创建好画布之后，就可以开始为其添加组件了。

9.4　小结

我们的场景已经准备好了，可以开始实现游戏玩法所需要的系统了。第 10 章将深入太空深处，实现飞船的飞行控制系统。

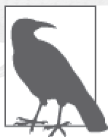
第10章

输入和飞行控制

在大体上对场景进行了布局后，就可以添加基本的游戏玩法了。在本章中，我们将开始构建一个让飞船在太空中移动的系统。

10.1 输入

这个游戏中使用两种不同的输入：一个虚拟摇杆，让玩家提供方向输入，用来确定飞行的方向；一个按钮，指示玩家是否想要发射飞船的激光束。



别忘了，恰当地测试触摸屏游戏输入的唯一方法是在触摸屏上进行测试。为了能够在不构建到设备的情况下测试游戏，需要使用 Unity Remote 应用（参见 5.1.1 节）。

10.1.1 添加摇杆

我们首先创建摇杆。摇杆由两个可见的组件构成：一个较大的方形控制区域，位于画布的左下角，以及一个较小的“手柄”，位于该方形控制区域的中心。当用户把手指放到该控制区域中时，摇杆将调整自己的位置，使手柄正位于手指的下方，并仍然处在中心位置。当手指移动时，手柄将随之移动。按照下面的步骤开始构建输入系统。

(1) **创建控制区域**。打开 GameObject 菜单，选择 UI → Panel。将新面板命名为 Joystick。首先将其设为方形，放到屏幕的左下角。将锚点设为 Lower Left。接下来，将该面板的宽度和高度均设为 250。

(2) **向控制区域添加图片**。将 Image 组件的 Source Image 设置改为 `Pad` 精灵。

完成之后，面板应该如图 10-1 所示。

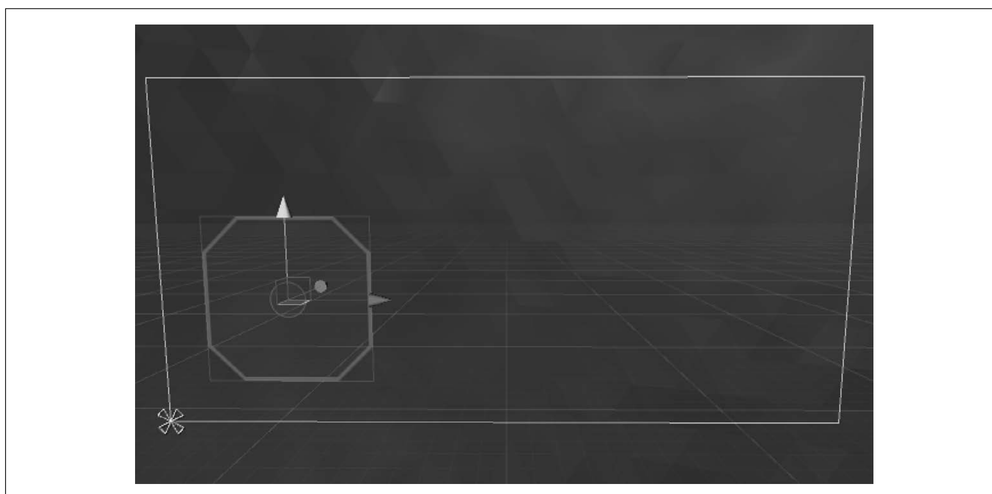


图 10-1：摇杆控制区域

(3) **创建手柄**。创建另一个 Panel UI 对象，命名为 Thumb。

将手柄设为 Joystick 的子对象，将其锚点设为 Middle Center，宽度和高度均设为 80。将 *x* 和 *y* 位置设为 0。这将使手柄位于控制区域的中心。最后，将 Source Image 设为 `Thumb` 精灵。

(4) **添加 `VirtualJoystick` 脚本**。选择 Joystick，添加一个新的 C# 脚本，命名为 VirtualJoystick.cs。打开该文件，添加下面的代码：

```
//获取对Event接口的访问
using UnityEngine.EventSystems;

//获取对UI元素的访问
using UnityEngine.UI;

public class VirtualJoystick : MonoBehaviour,
  IBeginDragHandler, IDragHandler, IEndDragHandler {

  //被四处拖动的精灵
  public RectTransform thumb;

  //没有被拖动时，手柄和摇杆的位置
  private Vector2 originalPosition;
  private Vector2 originalThumbPosition;

  //将手柄从其原来的位置拖动了多远
  public Vector2 delta;

  void Start () {
    //当摇杆启动时，记录原始位置
    originalPosition
      = this.GetComponent<RectTransform>().localPosition;
    originalThumbPosition = thumb.localPosition;
```

```
    //禁用手柄，使其不可见
    thumb.gameObject.SetActive(false);

    //将delta重置为0
    delta = Vector2.zero;
  }

  //开始拖动时调用
  public void OnBeginDrag (PointerEventData eventData) {

    //使手柄可见
    thumb.gameObject.SetActive(true);

    //确定拖动发生在世界空间的什么位置
    Vector3 worldPoint = new Vector3();
    RectTransformUtility.ScreenPointToWorldPointInRectangle(
      this.transform as RectTransform,
      eventData.position,
      eventData.enterEventCamera,
      out worldPoint);

    //将摇杆置于该点
    this.GetComponent<RectTransform>().position
      = worldPoint;

    //确保手柄位于其相对于摇杆的起始位置
    thumb.localPosition = originalThumbPosition;
  }

  //发生拖动时调用
  public void OnDrag (PointerEventData eventData) {

    //确定现在拖动发生在世界空间的位置
    Vector3 worldPoint = new Vector3();
    RectTransformUtility.ScreenPointToWorldPointInRectangle(
      this.transform as RectTransform,
      eventData.position,
      eventData.enterEventCamera,
      out worldPoint);

    //将手柄置于该点
    thumb.position = worldPoint;

    //计算当前位置与起始位置的距离
    var size = GetComponent<RectTransform>().rect.size;

    delta = thumb.localPosition;

    delta.x /= size.x / 2.0f;
    delta.y /= size.y / 2.0f;

    delta.x = Mathf.Clamp(delta.x, -1.0f, 1.0f);
    delta.y = Mathf.Clamp(delta.y, -1.0f, 1.0f);
```

```
    }

    //拖动结束时调用
    public void OnEndDrag (PointerEventData eventData) {
      //重置摇杆的位置
      this.GetComponent<RectTransform>().localPosition
        = originalPosition;

      //将距离重置为0
      delta = Vector2.zero;

      //隐藏手柄
      thumb.gameObject.SetActive(false);
    }
  }
```

`VirtualJoystick` 类实现了 3 个关键的 C# 接口：`IBeginDragHandler`、`IDragHandler` 和 `IEndDragHandler`。当玩家开始拖动、持续拖动及结束拖动摇杆时，脚本将分别收到 `OnBeginDrag`、`OnDrag` 和 `OnEndDrag` 方法调用。这些方法接受一个参数，即 `PointerEventData` 对象，该对象包含了手指在屏幕上的位置信息，以及其他一些数据。

- 当拖动**开始**时，控制区域将调整自己的位置，使其中心点位于手指下方。
- 当拖动**持续**时，手柄将跟随手指移动，并保持在手指下方的位置。控制区域中心与手柄之间的距离将被计算出来，并存储到 `delta` 属性中。
- 当拖动**结束**时（即手指离开屏幕时），控制区域和手柄将重置回原来的位置。`delta` 属性将重置为 0。

完成输入系统构建，还需要下面的步骤。

(5) **配置摇杆**。选择 Joystick 对象，将 Thumb 对象拖动到 Thumb 框中。

(6) **测试摇杆**。运行游戏，在摇杆控制区域中单击并拖动。当拖动开始时，控制区域将会移动。随着你继续拖动，手柄也会移动。注意 Joystick 对象的 `delta` 值，你移动手柄时，这个值也应该变化。

10.1.2 输入管理器

设置好摇杆后，我们需要一种方法，让飞船从摇杆处获取信息，以便确定飞行方向。

我们**可以**把飞船直接连接到摇杆，但是这么做有一个问题。在游戏过程中，飞船会被摧毁，新的飞船将被创建。为了实现这一点，需要让飞船成为预设，这样游戏管理器就能够创建飞船的多个副本。但是，预设不能引用场景中的对象，这意味着新创建的飞船对象将无法引用摇杆。

更好的方法是创建一个 Input Manager 单例，使其始终在场景中，并引用摇杆。因为它不是从预设实例化的，所以不需要担心在创建时丢失引用。创建飞船时，飞船将（通过代码）使用 Input Manager 单例来访问摇杆，并获得摇杆的值。

(1) **创建 Singleton 代码**。在 Assets 文件夹中创建一个新的 C# 脚本，命名为 Singleton.cs。打开此文件，添加下面的代码：

```
//此类允许其他对象引用单个共享对象
//GameManager和InputManager使用此类

//为使用此类，需要进行继承，如：
// public class MyManager : Singleton<MyManager> {  }

//然后就可以访问此类的单个共享实例，如：
// MyManager.instance.DoSomething();

public class Singleton<T> : MonoBehaviour
  where T : MonoBehaviour {

  //此类的单个实例
  private static T _instance;

  //访问器。第一次调用时，将设置_instance
  //如果找不到合适的对象，将记录一个错误
  public static T instance {
    get {
      //如果还没有设置_instance……
      if (_instance == null)
      {
        //尝试找到该对象
        _instance = FindObjectOfType<T>();

        //如果找不到，就记录错误
        if (_instance == null) {
          Debug.LogError("Can’ t find "
            + typeof(T) + "!");
        }
      }

      //返回实例供使用
      return _instance;
    }
  }

}
```



Singleton 的代码与 *Gnome's Well* 中的 Singleton 代码是相同的。其功能描述参见 5.1.2 节的“创建单例类”。

(2) **创建 Input Manager**。创建一个新的空游戏对象，命名为 Input Manager。为其添加一个新的 C# 脚本，命名为 InputManager.cs。打开该文件，添加下面的代码：

```
public class InputManager : Singleton<InputManager> {

  //摇杆用于控制飞船方向
  public VirtualJoystick steering;

}
```

目前，`InputManager` 只是作为一个简单的数据对象：它只是存储对 `VirtualJoystick` 的引用。我们将在后面为其添加更多逻辑，以支持更多功能，例如发射飞船的武器。

(3) **配置 Input Manager**。将 Joystick 拖放到 Steering 框中。

设置好摇杆后，就可以使用它来控制飞船的飞行了。

10.2　飞行控制

目前，飞船只能向前移动。控制飞船的飞行，只需要改变其“前进”方向。为此，我们从虚拟摇杆获取信息，然后使用得到的信息更新飞船在太空中的方向。

在每一帧中，飞船使用摇杆指示的方向，以及控制飞船旋转速度的一个值，来生成一次新旋转。然后，将新旋转与飞船**当前**的方向合并起来，就得到飞船的新前进方向。

但是，我们不想让玩家横滚飞船，导致无法确定重要对象（如空间站）的位置。为了解决这个问题，飞行脚本还应用了一个**额外**的旋转，将飞船缓慢旋转到水平面上。这使得飞船就像在大气层中飞行的飞机一样，理解起来更加直观（但是就没那么符合现实情况了）。

(1) **添加 `ShipSteering` 脚本**。选择 Ship，然后添加一个新的 C# 脚本，命名为 ShipSteering.cs。打开该文件，添加下面的代码：

```
public class ShipSteering : MonoBehaviour {

  //飞船转动的速率
  public float turnRate = 6.0f;

  //飞船回归水平飞行的力度
  public float levelDamping = 1.0f;

  void Update () {

    //通过将摇杆的方向乘以turnRate，再将得到的值限制到半圆的90%，
    //创建一个新的旋转

    //首先，获取用户输入
    var steeringInput
      = InputManager.instance.steering.delta;

    //现在，创建一个向量作为旋转量
    var rotation = new Vector2();

    rotation.y = steeringInput.x;
    rotation.x = steeringInput.y;

    //乘以turnRate来得到想要控制的量
    rotation *= turnRate;

    //乘以半圆的90%来转换为弧度
    rotation.x = Mathf.Clamp(
      rotation.x, -Mathf.PI * 0.9f, Mathf.PI * 0.9f);
```

```
        //将弧度转换为旋转四元数
        var newOrientation = Quaternion.Euler(rotation);

        //将此旋转与当前的方向组合起来
        transform.rotation *= newOrientation;

        //接下来，使横滚程度为最小

        //首先确定，如果完全没有绕z轴横滚，方向将是什么
        var levelAngles = transform.eulerAngles;
        levelAngles.z = 0.0f;
        var levelOrientation = Quaternion.Euler(levelAngles);

        //将当前方向与此“零横滚”方向的一个很小的量进行组合；
        //发生在多个帧中时，对象将缓慢回归到零横滚
        transform.rotation = Quaternion.Slerp(
          transform.rotation, levelOrientation,
          levelDamping * Time.deltaTime);
    }
  }
```

`ShipSteering` 脚本使用摇杆输入来计算新的、平滑的旋转，然后将计算出的旋转应用到飞船。之后，脚本应用一个额外的轻微旋转，使飞船恢复水平飞行。

(2) **测试飞行**。启动游戏，飞船将开始向前移动。当你在摇杆区域单击并拖动时，飞船的方向将发生改变。使用这种方法可四处飞行。注意，如果飞船翻滚（例如，机头翘起，然后翻向一侧），飞船将试图回到水平状态。

10.2.1 指示器

因为本游戏涉及在 3D 空间中四处飞行，所以很容易弄不清游戏中各个对象的位置。空间站（最终）将遭受小行星的威胁，玩家需要知道空间站以及小行星的位置。

为了应对这个问题，我们将实现一个在屏幕上显示指示器的系统，以标明重要对象的位置。如果摄像机能够看到对象，那么对象的指示器将带有一个圆圈。如果对象在屏幕以外，那么它们的指示器将显示在屏幕边缘，指出其所在方向。

创建UI元素

首先，在画布中创建一个对象，作为所有指示器的容器。然后，构建一个指示器，并将其转换为预设，以便重用。具体设置步骤如下。

(1) **创建 Indicator 容器**。选择 Canvas，打开 GameObject 菜单，然后选择 Create Empty Child，创建一个新的空子对象。创建的新对象将具有一个 Rect Transform（用于 2D 对象，如画布元素），而不是常规的 Transform（用于 3D 元素）。将容器的锚点设为水平和垂直拉伸。

将新对象命名为 Indicators。

(2) **创建原型 Indicator**。打开 GameObject 菜单，选择 UI → Image，创建一个新的 Image。将新对象命名为 Position Indicator，设为前一步创建的 Indicators 对象的子对象。

将 Indicator 精灵拖放到精灵的 Source Image 框中。在 UI 文件夹可找到该精灵。

(3) **创建文本标签**。创建一个新的 Text 对象（同样，使用 GameObject 菜单的 UI 子菜单），设为 Position Indicator 精灵对象的子对象。

将文本的颜色改为白色，并将对齐方式设为水平和垂直居中。

将文本的 Text 设为 50m（在游戏过程中，文本将发生变化，但是这样设置能够方便你理解指示器看起来是什么样子的）。

将 Text 的锚点设为 center middle，并将其 *x* 和 *y* 位置设为 0。这将使文本在精灵中居中显示。

最后，为指示器使用自定义字体。在 Fonts 文件夹中找到 CRYSTAL-Regular 字体，将其拖放到 Text 的 Font 框。接下来，将 Font Size 改为 28。

完成之后，Text 组件的 Inspector 应该如图 10-2 所示，指示器对象应该如图 10-3 所示。

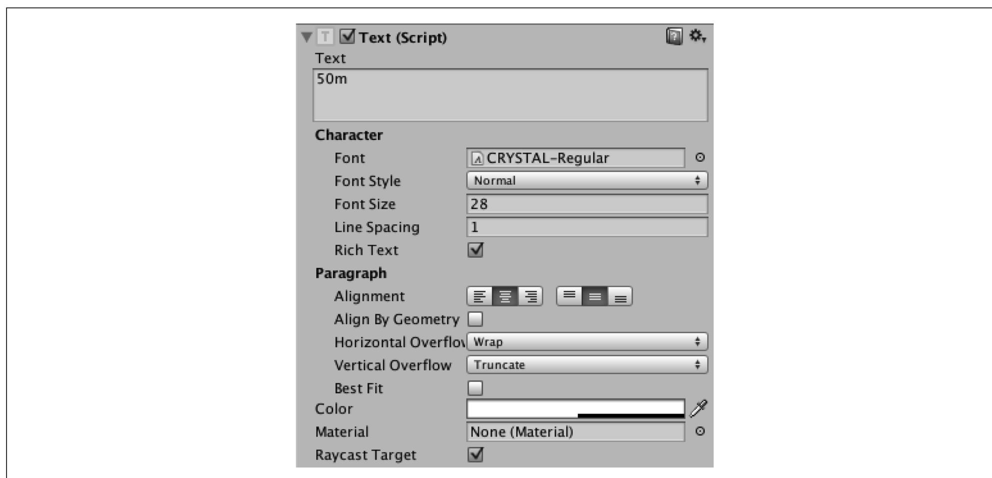


图 10-2：指示器文本标签的 Inspector

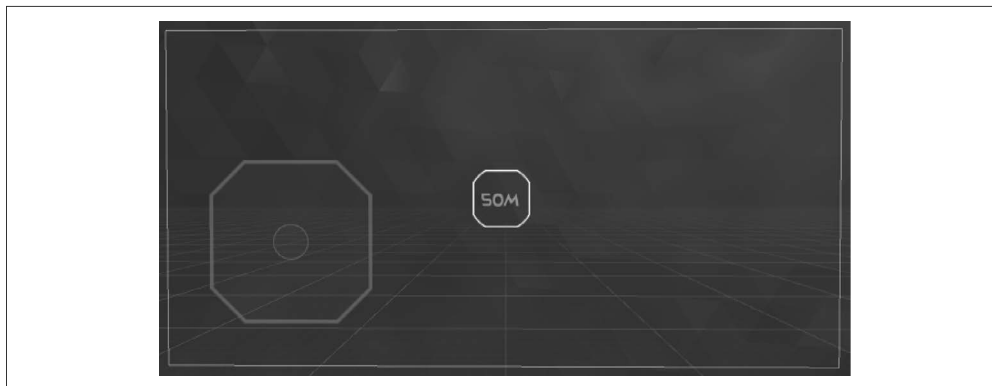


图 10-3：原型指示器

(4) **添加代码**。向原型 Indicator 对象添加一个新的 C# 脚本，命名为 Indicator.cs，并添加下面的代码：

```
//获得对UI类的访问
using UnityEngine.UI;

public class Indicator : MonoBehaviour {

  //我们跟踪的对象
  public Transform target;

  //测量从目标到此变换的距离
  public Transform showDistanceTo;

  //显示测量出的距离的标签
  public Text distanceLabel;

  //应该距离画面边缘多远
  public int margin = 50;

  // 图片的着色
  public Color color {
    set {
      GetComponent<Image>().color = value;
    }
    get {
      return GetComponent<Image>().color;
    }
  }

  //设置指示器
  void Start() {
    //隐藏标签；如果有了目标，会在Update方法中重新启用
    distanceLabel.enabled = false;

    //在启动时，等待一帧后显示，以避免出现视觉上的错误
    GetComponent<Image>().enabled = false;

  }

  //每一帧更新指示器的位置
  void Update()
  {

    //目标消失了吗？那么我们也应该离开了
    if (target == null) {
      Destroy (gameObject);
      return;
    }

    //如果有一个目标来计算距离，就计算距离并显示在distanceLable中
    if (showDistanceTo != null) {

      //显示标签
      distanceLabel.enabled = true;

      //计算距离
      var distance = (int)Vector3.Magnitude(
```

```
        showDistanceTo.position - target.position);

      //在标签中显示距离
      distanceLabel.text = distance.ToString() + "m";
    } else {
      //不显示标签
      distanceLabel.enabled = false;
    }

    GetComponent<Image>().enabled = true;

    //计算出对象在屏幕空间中的位置
    var viewportPoint =
      Camera.main.WorldToViewportPoint(target.position);

    //这个点位于我们身后吗?
    if (viewportPoint.z < 0) {
      //将其推到画面边缘
      viewportPoint.z = 0;
      viewportPoint = viewportPoint.normalized;
      viewportPoint.x *= -Mathf.Infinity;
    }

    //计算出我们应该在视口空间的什么位置
    var screenPoint =
      Camera.main.ViewportToScreenPoint(viewportPoint);

    //限制到画面的边缘
    screenPoint.x = Mathf.Clamp(
      screenPoint.x,
      margin,
      Screen.width - margin * 2);

    screenPoint.y = Mathf.Clamp(
      screenPoint.y,
      margin,
      Screen.height - margin * 2);

    //计算出视口空间坐标在画布空间中的位置
    var localPosition = new Vector2();
    RectTransformUtility.ScreenPointToLocalPointInRectangle(
      transform.parent.GetComponent<RectTransform>(),
      screenPoint,
      Camera.main,
      out localPosition);

    //更新位置
    var rectTransform = GetComponent<RectTransform>();
    rectTransform.localPosition = localPosition;

  }
}
```

指示器代码的工作方式如下。

- 在每一帧中，`Update` 方法把指示器跟踪对象的 3D 坐标转换到**视口空间**（viewport space）。在视口空间中，坐标代表屏幕上的位置，(0, 0, 0) 是屏幕的左下角，(1, 1, 0) 是屏幕的右上角。视口空间坐标的 *z* 分量代表（按照世界单位）距离摄像机多远。

  这意味着很容易判断某个对象在不在屏幕上，或者位于玩家的前方还是后方。如果某个对象视口空间坐标的 *x* 和 *y* 分量不在 (0, 0) 到 (1, 1) 之间，那么它在屏幕之外。如果其 *z* 坐标小于 0，那么它在玩家后方。
- 如果对象在玩家后方（即 *z* 坐标小于 0），那么需要把标记移动到一侧。否则，位于玩家后方对象的指示器将出现在屏幕中央，这会让玩家误以为该指示器跟踪的对象位于自己前方。

  为使标记移动到一侧，将视口的 *x* 分量乘以负无穷大。通过乘以无穷大，指示器将始终位于屏幕的最左端或最右端。乘以**负**无穷大是为了补偿对象位于玩家身后这一点。
- 接下来，将视口空间坐标转换到屏幕空间，然后固定，使其不会位于窗口以外。此外，还使用了一个边距参数，使指示器更向内一些，以确保显示距离的文本标签始终是可读的。
- 最后，将屏幕空间坐标转换到指示器容器的坐标空间，并用来更新指示器的位置。之后，指示器就位于正确的位置。

指示器还会自己进行清理：每一帧中，它们都会检查自己的目标是否为 `null`；如果是，就销毁自己。

设置指示器还剩最后一步。完成之后，就可以把这个原型指示器转换为预设。

(5) **连接距离标签**。将 Text 子对象拖放到 Distance Label 框中。

(6) **将原型转换为预设**。将 Position Indicator 对象拖放到 Project 窗格中，会创建一个新的预设对象，从而可在运行时创建多个 Indicator。

创建了这个预设之后，从场景中删除原型。

10.2.2 Indicator Manager

Indicator Manager 是一个单例对象，管理着创建指示器的过程。其他任何需要在屏幕上添加指示器的对象都将用到这个对象，特别是空间站和小行星。

将 Indicator Manager 设为单例，就可以在场景中创建和设置该对象，不需要任何特殊操作来让加载自预设的对象知道管理器的存在。

(1) **创建 Indicator Manager**。创建一个新的空对象，命名为 Indicator Manager。

(2) **添加 `IndicatorManager` 脚本**。向对象添加一个新的 C# 脚本，命名为 IndicatorManager.cs，在其中添加下面的代码：

```
using UnityEngine.UI;

public class IndicatorManager : Singleton<IndicatorManager> {

  //所有指示器都是此对象的子对象
  public RectTransform labelContainer;
```

```
    //为每个指示器实例化的预设
    public Indicator indicatorPrefab;

    //其他对象将调用此方法
    public Indicator AddIndicator(GameObject target,
      Color color, Sprite sprite = null) {

      //创建标签对象
      var newIndicator = Instantiate(indicatorPrefab);

      //使其跟踪目标
      newIndicator.target = target.transform;

      //更新其颜色
      newIndicator.color = color;

      //如果收到精灵，就将指示器的精灵设置为收到的精灵
      if (sprite != null) {
        newIndicator
          .GetComponent<Image>().sprite = sprite;
      }

      //添加到容器中
      newIndicator.transform.SetParent(labelContainer, false);

      return newIndicator;
    }

  }
```

Indicator Manager 只提供了一个 `AddIndicator` 方法，该方法实例化一个 Indicator 预设，使用要跟踪的目标对象和要为精灵着色的颜色配置该预设，然后把它添加到指示器的容器中。如果想创建一个特殊的指示器，也可以选择为这个方法提供自己的 `Sprite`（后面在添加飞船的目标标线时也会这么做）。

写好了 `IndicatorManager` 的源代码后，就需要配置 Indicator Manager。管理器需要知道两条信息：为指示器实例化哪个预设，以及哪个对象是指示器的父对象。

(3) **设置 Indicator Manager**。将 Indicators 容器对象拖放到 Label Container 框中，将 Position Indicator 预设拖动到 Indicator Prefab 框中。

接下来为空间站添加代码，当空间站启动时添加一个指示器。

(4) **选择空间站**。

(5) **为其添加 `SpaceStation` 脚本**。为空间站对象添加一个新的 C# 脚本，命名为 SpaceStation.cs，并添加下面的代码：

```
  public class SpaceStation : MonoBehaviour {

    void Start () {
      IndicatorManager.instance.AddIndicator(
        gameObject,
        Color.green
```

```
        );
      }

    }
```

以上代码让 `IndicatorManager` 单例添加一个新的指示器来跟踪此对象，并且将指示器设置为绿色。

(6) **运行游戏**。现在空间站将附加一个指示器。

现在还不会显示距离，因为空间站没有设置 `showDistanceTo` 变量。这是有意而为之：我们将为 Asteroids 设置该变量，但是不会为空间站设置。如果屏幕上的数字太多，会让人感到混乱。

10.3 小结

祝贺你！你几乎是完全从头开始构建了一个空战游戏。第 11 章将扩展这个游戏，并添加实际的游戏玩法。

第11章

添加武器及锁定目标

有了一个能够四处飞行的飞船后，就可以添加更多的游戏玩法了。我们首先为飞船添加武器，之后将确定一个射击目标。

11.1 武器

飞船在每次发射武器时会发出一个激光束，它会一直前进，直到击中对象或者生存期结束。如果击中对象，并且被击中的对象会受到伤害，那么激光束需要向该对象传递信息。

我们可以创建一个对象，为其添加一个碰撞器，并让它以一定的速率向前移动（与飞船类似）。武器出现的方式有多种不同的选择：可以创建 3D 导弹模型、创建粒子效果，或创建一个精灵。具体细节由你决定，并不会影响武器在游戏中的实际行为。

本章中将使用**轨迹渲染器**来显示武器。当武器移动时，轨迹渲染器会在后面创建一条轨迹，轨迹最终会慢慢消失。因此，轨迹渲染器特别适合表现移动的对象，例如摇摆的物体和飞行的炮弹。

武器的轨迹渲染器很简单：它留下一条细红线，这条线会随着时间变得越来越细。因为武器总是向前移动的，所以这将创造出一种很好看的强烈激光束的效果。

武器的非图形组件将由一个**运动学刚体**实现。一般情况下，刚体会响应自己受到的力：重力将刚体向下拉，而当刚体被另外一个刚体撞击时，牛顿第一运动定律意味着刚体的速度会发生改变。但是，我们不想让武器被撞到一边。通过告诉 Unity，某个刚体无视自己受到的任何力，同时仍然让该刚体与其他刚体发生碰撞，我们就使其具有**运动学**特性。



为什么要为武器使用刚体？这是一个很合理的问题。毕竟，飞船没有使用刚体，为什么武器要使用？

原因在于 Unity 物理引擎的局限。仅当发生碰撞的对象中至少有一个有刚体时，碰撞才能发生。因此，为了确保当武器接触到另外一个对象时总是能够得到通知，我们为其附加一个刚体，并使其具有运动学特性。

首先创建武器对象，并设置其碰撞属性。然后，添加 `Shot` 代码，将武器设置为以恒定速度向前飞行。

(1) **创建武器**。创建一个新的空游戏对象，命名为 Shot。

为对象添加一个刚体组件。然后，确保关闭 Use Gravity，并打开 Is Kinematic。

为对象添加一个球体碰撞器。将其半径设为 0.5，并确保其中心是 (0, 0, 0)。打开 Is Trigger 设置。

(2) **添加 `Shot` 脚本**。为对象添加一个新的 C# 脚本，命名为 `Shot`。打开 Shot.cs，添加下面的代码：

```
//以特定速度向前移动，并在特定时间后消失
public class Shot : MonoBehaviour {

  //武器向前移动的速度
  public float speed = 100.0f;

  //在这么多秒后移除此对象
  public float life = 5.0f;

  void Start() {
    //在life秒后销毁
    Destroy(gameObject, life);
  }

  void Update () {
    //以恒定速度向前移动
    transform.Translate(
      Vector3.forward * speed * Time.deltaTime);
  }
}
```

`Shot` 代码极其简单，关注两个任务：确保武器在一段时间后消失，以及使武器一直向前移动。

在调用 `Destroy` 方法时一般只使用一个参数，即想要从游戏中移除的对象。但是，也可以传入一个可选的参数，即从调用时算起，让对象在多少秒以后销毁。`Start` 方法调用了 `Destroy` 方法并传入了 `life` 变量，告诉 Unity 在经过 `life` 秒后销毁对象。

`Update` 函数只是使用 `transform` 的 `Translate` 方法，让对象以恒定速度向前移动。通过把 `Vector3.forward` 属性乘以 `speed` 和 `Time.deltaTime`，对象在每一帧中都将以恒定速度向前移动。

接下来将添加武器的图形。如前所述，我们将使用轨迹渲染器来创建武器的视觉效果。轨迹渲染器使用材质来定义轨迹的外观，这意味着我们需要创建一个材质。

材质可以是任何样子，但是为了保持这个游戏的简洁观感，我们将采用无光照的纯红色。

(1) **创建一个新材质**。命名为 Shot。

(2) **更新着色器**。为了使轨迹显示为纯色，不带任何光照，需要将材质的着色器设为 Unlit/Color。

(3) **设置颜色**。改变了材质的着色器后，材质的参数将变为只有一个参数，即要使用的颜色。将颜色改为好看的亮红色。

创建材质后，就可以在轨迹渲染器中使用。

(1) **创建 Shot 的图形对象**。创建一个新的空游戏对象，命名为 Graphics，设为 Shot 对象的子对象，位置为 (0, 0, 0)。

(2) **创建轨迹渲染器**。向 Graphics 对象添加一个新的 Trail Renderer 组件。

添加后，Cast Shadows、Receive Shadows 和 Use Light Probes 都将被关闭。

接下来，将 Time 设为 0.05，Width 设为 0.2。

(3) **使 trail 末端逐渐变细**。双击 Width 字段下方的曲线视图，将显示一个新的控制点。拖动该控制点到曲线视图的右下角。

(4) **应用 Shot 材质**。打开 Materials 列表，拖入刚刚创建的 Shot 材质。

完成后，轨迹渲染器的 Inspector 应该如图 11-1 所示。

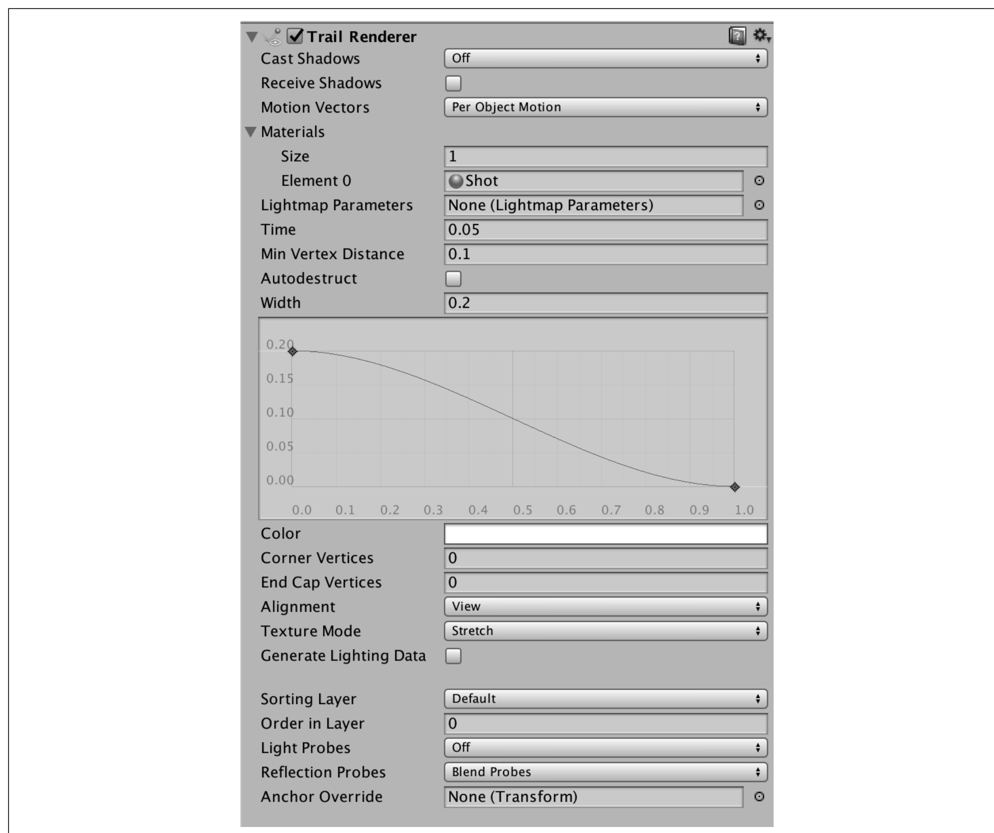


图 11-1：为武器配置的轨迹渲染器



Shot 对象还没有创建好：目前还没有办法测试飞船的武器发射，不过我们很快就将添加该功能。

还剩最后一步：使 Shot 成为预设。

(1) **将 Shot 对象从场景中拖放到 Objects 文件夹**。这将把 Shot 转换为预设。
(2) **从场景中删除 Shot**。

接下来将创建一个对象来处理武器的发射。

11.1.1 飞船的武器

当玩家想要开始发射飞船的激光时，我们需要有东西来实际创建 Shot 对象。飞船发射激光时，并不是简单地在每次触摸 Fire 按钮时生成一个 Shot，而是更复杂一些。我们想要的效果是，当按住 Fire 按钮时，飞船将以固定速率发射 Shot 对象。

另外，我们需要指定从**什么位置**发射武器。飞船的概念图（如图 9-3 所示）显示，飞船的两翼装有激光炮，所以武器应该来自这两个位置。

关于武器的**发射方式**，需要做一个决定。可以同时发射两个武器，也可以交替发射——先从左侧发射，再从右侧发射。在这个游戏中，我们决定采用交替模式，因为这样会让发射动作看起来是连贯的。但是，你可以有自己的决定。试试不同的发射模式，看看它们会如何改变飞船给你的感觉。

`ShipWeapons` 脚本负责处理武器的发射。此脚本使用前一节创建的武器预设，以及一个 `Transform` 对象数组。当武器开始发射时，会轮流在每个 `Transform` 对象的位置实例化武器。当到达 `Transform` 数组末尾时，会返回其开始位置。

(1) **向飞船添加 `ShipWeapons` 脚本**。选择 Ship，添加一个新的 C# 脚本，命名为 ShipWeapons.cs，并添加下面的代码：

```
public class ShipWeapons : MonoBehaviour {

  //为每个武器使用的预设
  public GameObject shotPrefab;

  //武器发射位置的列表
  public Transform[] firePoints;

  //发射武器的firePoints的索引
   Private int firePointIndex;

  //由InputManager调用
  public void Fire() {

    //如果没有武器发射点，就返回
    if (firePoints.Length == 0)
      return;
```

```
      //计算出从哪个发射点发射
      var firePointToUse = firePoints[firePointIndex];

      //在发射点位置使用其旋转，创建新武器
      Instantiate(shotPrefab,
        firePointToUse.position,
        firePointToUse.rotation);

      //移动到下一个发射点
      firePointIndex++;

      //如果在到达列表中最后一个发射点以后继续移动，
      //就回到队列的开始位置
      if (firePointIndex >= firePoints.Length)
        firePointIndex = 0;

    }

  }
```

`ShipWeapons` 脚本跟踪武器应该出现的位置的一个列表（`firePoints` 变量），以及代表每个武器的一个预设（`shotPrefab` 变量）。另外，它还会跟踪下一个武器应该来自**哪个**发射点（`firePointIndex` 变量）。当按下 `Fire` 按钮时，从一个发射器射击一个武器，然后更新 `firePointIndex`，使其引用下一个发射点。

(2) **创建武器发射点**。创建一个新的空游戏对象，命名为 Fire Point 1，设为 Ship 对象的子对象，然后按 Ctrl-D 键复制它（在 Mac 上为 Command-D 键）。这将创建另外一个名为 Fire Point 1 的空对象。

将 Fire Point 1 的位置设为 (−1.9, 0, 0)。这将使其位于飞船的左侧。

将 Fire Point 2 的位置设为 (1.9, 0, 0)。这将使其为飞船的右侧。

完成之后，Fire Point 1 和 Fire Point 2 的位置应该分别如图 11-2 和图 11-3 所示。

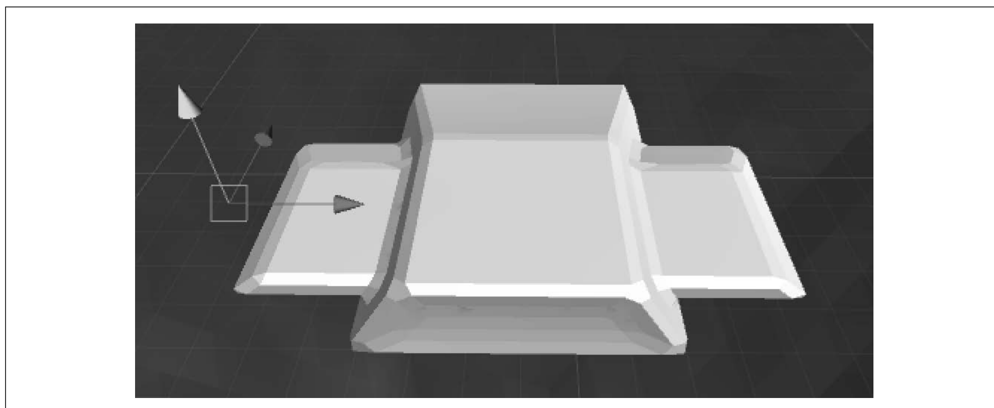


图 11-2：Fire Point 1 的位置

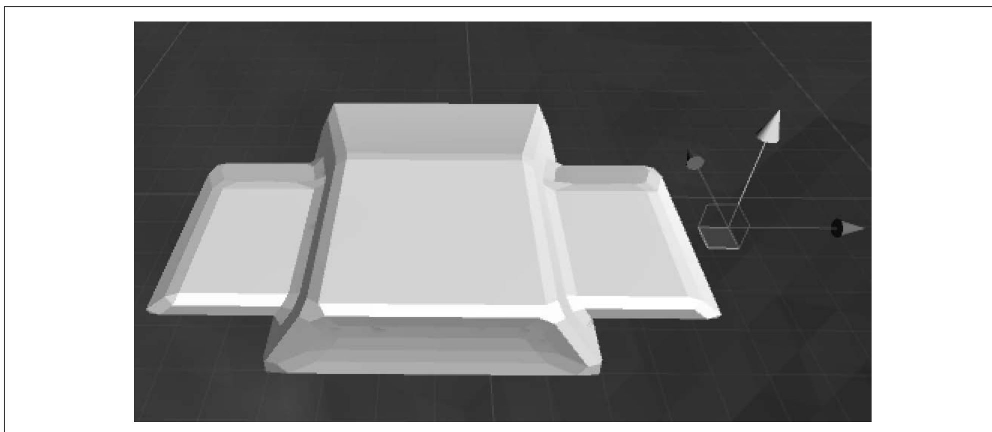


图 11-3：Fire Point 2 的位置

(3) **配置 `ShipWeapons` 脚本**。将前一节创建的 Shot 预设拖放到 `ShipWeapons` 的 Shot Prefab 框中。

接下来，需要将两个 Fire Point 对象都添加到 `ShipWeapons` 脚本中。为此，可以将 FirePoints 数组的大小设为 2，然后每次拖入一个对象，但是还有一种更快的方法。

选择 Ship，然后单击 Inspector 右上角的锁。这将锁定 Inspector，意味着当选择另外一个对象时，此 Inspector 对应的对象不会改变。

接下来，在 Hierarchy 中，单击 Fire Point 1，然后按住 Ctrl 键（Mac 上为 Command 键）并单击 Fire Point 2，同时选中两个 Fire Point 对象。

然后，将这两个对象拖放到 `ShipWeapons` 的 Fire Points 框中。确保把它们拖动到文本 Fire Points 上（而不是该文本下方的任何地方），否则不会起效果。



这种方法适合脚本中的**任意**数组变量，可以省去**大量**的拖放操作。有一点要记住：当从 Hierarchy 拖放到数组中时，对象的顺序不一定保留。

(4) **解锁 Inspector**。完成对 `ShipWeapons` 脚本的配置后，单击右上角的锁图标解锁 Inspector。



在本游戏中，飞船只有两个发射点，但是脚本能够处理更多的发射点。如果你愿意，可以添加更多的发射点，只是要确保它们是 Ship 对象的子对象，并且把它们添加到 Inspector 的 Fire Points 列表中。

接下来将在游戏的界面中添加一个 Fire 按钮，用于实际发射武器。

11.1.2 Fire按钮

现在，我们将添加一个“开火”按钮。当用户开始触摸该按钮时，飞船将开始发射武器；当用户的手指离开屏幕时，将停止发射武器。

游戏中将只有一个 Fire 按钮，但是会有多个飞船。这意味着我们不能把 Fire 按钮直接挂钩到飞船。我们需要添加对 Input Manager 的支持，以允许其处理 ShipWeapons 脚本的多个实例。

Input Manager 处理 ShipWeapons 脚本实例的方式如下。因为游戏中每次只出现一个飞船，游戏中每次也只会有一个 ShipWeaspons 实例。ShipWeapons 脚本出现时将联系 InputManager 单例，并通知它：自己是当前的 ShipWeapons 脚本。InputManager 将记录下这一点，并在发射系统中使用该脚本。

最后，Fire 按钮将连接到 Input Manager 对象，并且当 Fire 按钮被按下时发送一个“开火”消息，当按钮被松开时发送一个“停止射击”消息。Input Manager 将把这些消息转发给当前的 ShipWeapons 脚本，从而实现射击。



实现上述目的的另一种方法是使用 FindObjectOfType 方法。该方法在全部对象中搜索与一个类型匹配的任意组件，然后返回找到的第一个组件。使用 FindObjectOfType 就不需要让对象把自己**注册**为当前的对象，但这样做是有代价的：FindObjectOfType 方法很慢，因为它需要检查场景中每个对象的每个组件。偶尔使用无伤大雅，但是不应该在每一帧中都使用这个方法。

首先，我们在 InputManager 类中添加代码来跟踪当前的 ShipWeapons 实例，然后在 ShipWeapons 中添加代码，使其出现时注册为当前实例，而当组件被移除时（例如飞船被摧毁时）取消注册。

我们需要在 InputManager 中添加 ShipWeapons 管理代码。具体来说，就是在 InputManager 类中添加下面的属性和方法：

```
public class InputManager : Singleton<InputManager> {

  //用来控制飞船方向的摇杆
  public VirtualJoystick steering;

> //发射武器的间隔，单位为秒
> public float fireRate = 0.2f;
>
> //当前发射武器的ShipWeapons脚本
> private ShipWeapons currentWeapons;
>
> //如果为true，那么正在发射武器
> private bool isFiring = false;
>
> //由ShipWeapons调用来更新currentWeapons变量
> public void SetWeapons(ShipWeapons weapons) {
>   this.currentWeapons = weapons;
> }
>
> //类似地，调用此方法来重置currentWeapons变量
> public void RemoveWeapons(ShipWeapons weapons) {
```

```
>
>     //如果currentWeapons对象为weapons，则将其设为null
>     if (this.currentWeapons == weapons) {
>       this.currentWeapons = null;
>     }
>   }
>
>   //当用户开始触摸Fire按钮时调用
>   public void StartFiring() {
>
>     //启动开始发射武器的例程
>     StartCoroutine(FireWeapons());
>   }
>
>   IEnumerator FireWeapons() {
>
>     //标记为正在发射武器
>     isFiring = true;
>
>     //只要isFiring为true，就循环
>     while (isFiring) {
>
>       //如果有一个weapons脚本，就告诉它发射武器
>       if (this.currentWeapons != null) {
>         currentWeapons.Fire();
>       }
>
>       //等待fireRate秒，然后再次发射
>       yield return new WaitForSeconds(fireRate);
>
>     }
>
>   }
>
>   //当用户停止触摸Fire按钮时调用
>   public void StopFiring() {
>
>     //将此变量设为false将停止FireWeapons中的循环
>     isFiring = false;
>   }
>
  }
```

这段代码跟踪当前负责从飞船发射武器的 ShipWeapons 脚本。创建和销毁武器时，ShipWeapons 脚本将分别调用 SetWeapons 和 RemoveWeapons 方法。

当调用 StartFiring 方法时，将启动一个新的协程，后者通过调用 ShipWeapons 组件的 Fire 方法发射武器，然后等待 fireRate 秒。当 isFiring 为 true 时，将循环这个过程。当调用 StopFiring 方法时，将 isFiring 设为 false。当用户开始触摸和结束触摸 Fire 按钮时，将分别调用 StartFiring 和 StopFiring 方法，我们稍后就进行相关设置。

接下来需要在 ShipWeapons 中添加与 InputManager 通信的代码。为此，在 ShipWeapons 类中添加下面的方法：

```
public class ShipWeapons : MonoBehaviour {

  //用于每发武器的预设
  public GameObject shotPrefab;

>   public void Awake() {
>     //当此对象启动时，告诉InputManager使用自己作为当前的武器对象
>     InputManager.instance.SetWeapons(this);
>   }
>
>   //移除对象时调用
>   public void OnDestroy() {
>     //如果没有在玩游戏，就不做此处理
>     if (Application.isPlaying == true) {
>       InputManager.instance
>         .RemoveWeapons(this);
>     }
>   }

    //武器发射位置的列表
    public Transform[] firePoints;

    //发射武器的firePoints的索引
    private int firePointIndex;

    //由InputManager调用
    public void Fire() {

      //如果没有武器发射点，就返回
      if (firePoints.Length == 0)
        return;

      //计算出从哪个发射点发射
      var firePointToUse = firePoints[firePointIndex];

      //在发射点位置使用其旋转，创建新武器
      Instantiate(shotPrefab,
        firePointToUse.position,
        firePointToUse.rotation);

      //移动到下一个发射点
      firePointIndex++;

      //如果在到达列表中最后一个发射点以后继续移动，
      //就回到队列的开始位置
      if (firePointIndex >= firePoints.Length)
        firePointIndex = 0;

    }

  }
```

创建一个飞船后，`ShipWeapons` 脚本的 `Awake` 方法将访问 `InputManager` 单例，并将自身注册为当前的武器脚本。当销毁脚本时（例如我们稍后添加的飞船与小行星碰撞的情形），`OnDestroy` 方法将使输入管理器注销此脚本。



注意，`OnDestroy` 方法在继续设置之前，会检查 `Application.isPlaying` 是否为 `true`。这是因为，在编辑器中停止玩一个游戏时，所有的对象都会被销毁。其结果是，所有具有 `OnDestroy` 方法的脚本，其 `OnDestroy` 方法都会被调用。但是，这产生了一个问题：请求 `InputManager.singleton` 会引发错误，因为游戏正在结束，该对象已被销毁。

为了避免这个问题，我们检查 `Application.isPlaying`。要求 Unity 停止游戏后，该属性将变为 `false`，这就完全避免了对 `InputManager.singleton` 的错误调用。

现在来创建 Fire 按钮，通知 Input Manager 启动和停止发射。默认情况下，只有在“单击”（手指按下并松开）之后，按钮才会发送一个消息。因为我们需要告诉 Input Manager 按钮被按下和松开的操作，所以不能使用默认的按钮行为。相反，我们需要使用 Event Trigger，当发生 Pointer Down 和 Pointer Up 事件时都单独发送消息。

我们首先来创建按钮，并设置其位置。打开 GameObject 菜单，选择 UI → Button，创建一个新按钮，命名为 Fire Button。

单击 Inspector 左上角的 Anchor 按钮，在按住 Alt 键（Mac 上为 Option 键）的同时单击 Bottom Right 选项，将按钮的锚点和轴点设置为 Bottom Right。

接下来，将按钮的位置设为 (−50, 50, 0)，这将把按钮放到画布的右下角。将按钮的宽度和高度均设为 160。

将按钮的 Image 组件的 Source Image 设为 `Button` 精灵。将 Image Type 设为 `Sliced`。

选择 Fire 按钮的 Text 子对象，将其文本设为 Fire。设置 Font 为 CRYSTAL-Regular，Font Size 为 28。设置对齐方式为垂直和水平居中。

最后，单击 Color 字段，然后在 Hex Color 字段中，输入 3DFFD0FF，将 Fire 按钮的颜色设为淡蓝绿色，如图 11-4 所示。

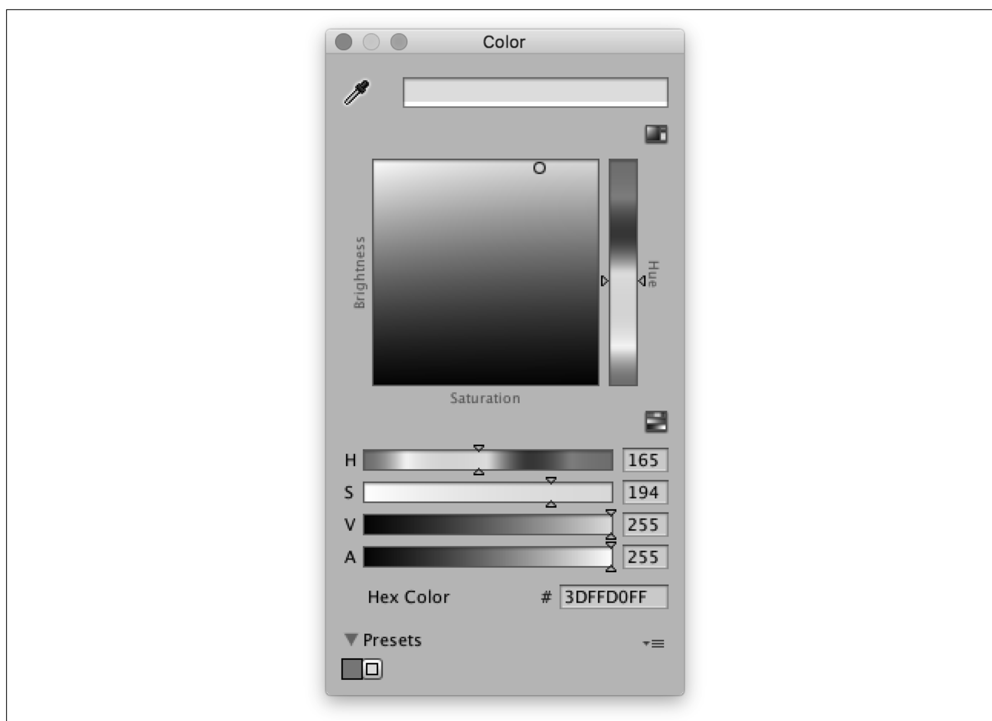


图 11-4：设置 Fire 按钮的标签颜色

完成之后，按钮应该如图 11-5 所示。

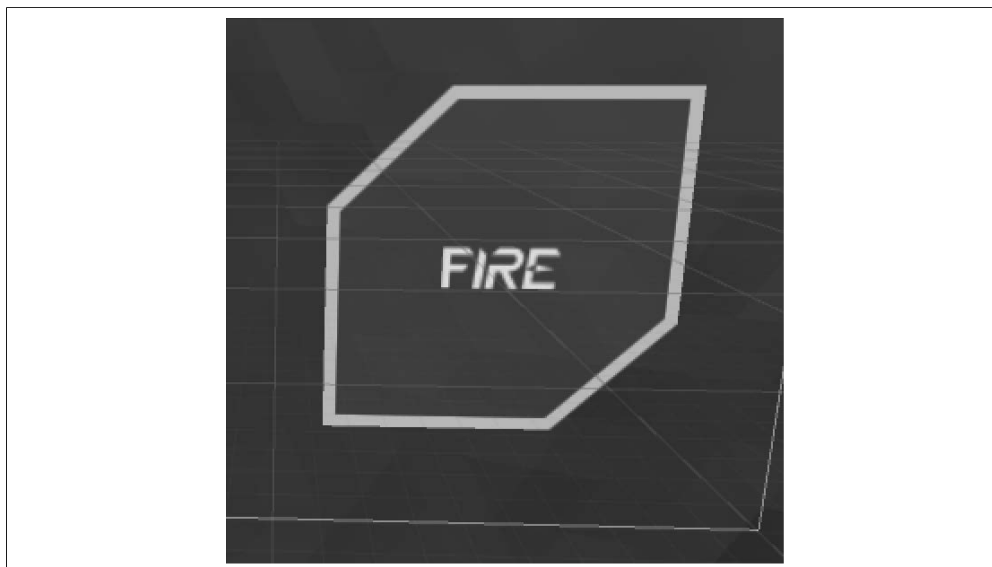


图 11-5：Fire 按钮（另见彩插）

接下来，我们根据自己的需要来设置按钮的行为。

(1) **移除 Button 组件**。选择 Fire Button 对象，然后单击 Button 组件右上角的设置图片。单击 Remove Component 按钮。

(2) **添加一个 Event Trigger，并添加 Pointer Down 事件**。添加一个新的 Event Trigger 组件，然后单击 Add Event Type 按钮。从显示的菜单中选择 PointerDown。

列表中将出现一个新的事件，包含当触控点触摸到按钮内部（即用户开始触摸 Fire 按钮）时运行的对象和方法的列表。默认情况下这个列表为空，所以需要添加一个新的目标。

(3) **配置 Pointer Down 事件**。单击 PointerDown 列表底部的 + 按钮，列表中将出现一个新项。

从 Hierarchy 面板中拖放 Input Manger 对象到该框中。接下来，将方法从 No Function 改为 InputManager → StartFiring。

(4) **添加并配置 Pointer Up 事件**。接下来，需要为手指离开屏幕的操作添加一个事件。再次单击 Add Event Type，然后选择 PointerUp。

按照配置 PointerDown 的方式配置此事件，但是这次应该调用 InputManager 的 StopFiring 方法。

完成之后，Inspector 应该如图 11-6 所示。

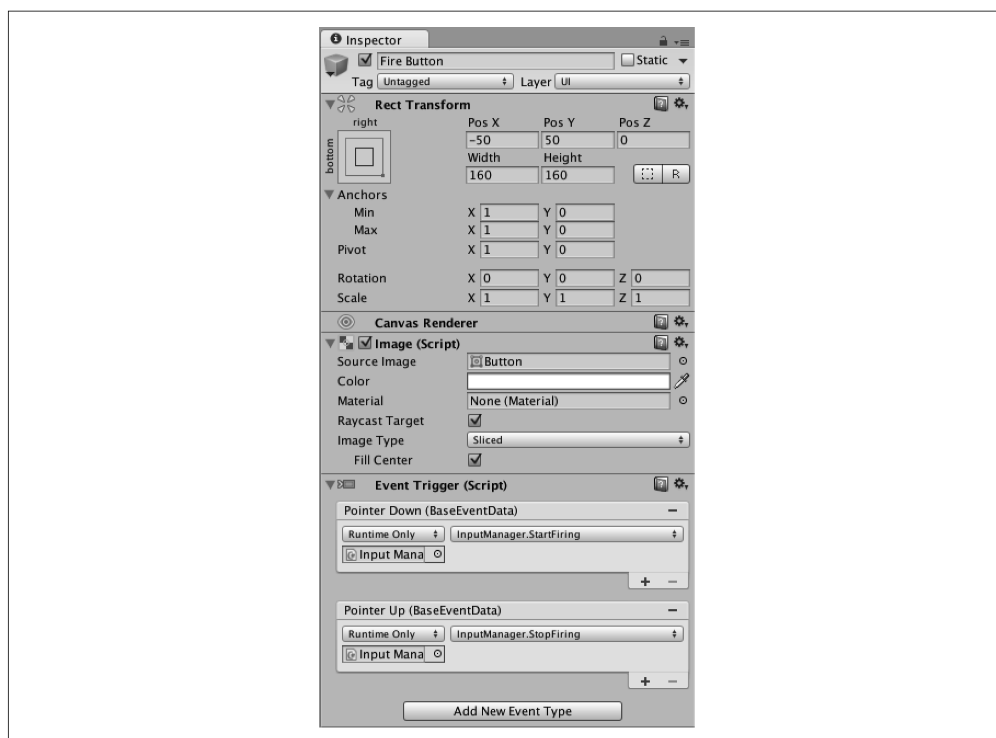


图 11-6：配置好的 Fire 按钮

(5) **测试 Fire 按钮**。玩游戏。按下 Fire 按钮时，将发射武器！

11.2 目标标线

目前，还没有明确的方法让用户知道他们在瞄准什么地方。由于摄像机和飞船都可能在移动，瞄准实际上是很困难的。为了解决这个问题，我们将使用前面创建的指示器系统，在屏幕上显示一个目标标线。

我们将创建一个新对象。与 Space Station 一样，这个对象将告诉 Indicator Manager 在屏幕上创建一个新的指示器，用于跟踪自己的位置。此对象是飞船的一个不可见子对象，位于距离飞船前端一定距离处。这会让指示器放到玩家当前瞄准的位置。

最后，这个指示器应该使用一个特殊图标，以便清楚地表明它代表的是瞄准点。Target Reticle.psd 图片包含一个十字准星图标，能够很好地满足这个要求。

(1) **创建 Target 对象**。将新对象命名为 Target，并设为 Ship 的子对象。
(2) **设置 Target 的位置**。将 Target 对象的位置设为 (0, 0, 100)。这将使目标位于飞船前方一定距离处。
(3) **添加 `ShipTarget` 脚本**。为 Target 对象添加一个新的 C# 脚本，命名为 ShipTarget.cs，并添加下面的代码：

```
public class ShipTarget : MonoBehaviour {

  //目标标线使用的精灵
  public Sprite targetImage;

  void Start () {

    //使用黄色和自定义精灵注册一个新指示器，
    //用于跟踪此对象
    IndicatorManager.instance.AddIndicator(gameObject,
      Color.yellow, targetImage);
  }

}
```

`ShipTarget` 代码使用 `targetImage` 变量，告诉 Indicator Manager 在屏幕上使用一个自定义精灵。这意味着需要配置 Target Image 框。

(4) **配置 `ShipTarget` 脚本**。将 Target Reticle 精灵拖放到 `ShipTarget` 脚本的 Target Image 框中。
(5) **玩游戏**。在四处飞行时，飞船瞄准的位置将显示一个目标标线。

11.3 小结

现在我们就准备好了武器系统。你应该试着驾驶飞船，看看它给你什么感觉。你可能注意到，太空中没有可供射击的目标。虽然大家都知道，太空中空空荡荡，也没有什么东西，可以说我们很大程度上真实地表现了太空，但是对于游戏来说，这并不有趣。翻到下一页，我们来解决这个问题。

第12章

小行星与伤害

12.1 小行星

现在，我们已经有了一个在太空中飞行的飞船，在屏幕上也有了指示器，我们也能够瞄准和发射激光炮。但是，我们**还没有**一个能够射击的有效目标（空间站不算）。

现在是时候解决这个问题了。我们将创建小行星，它们本身除了四处漂浮以外并不会做什么。另外，我们还将创建一个系统，用于创建这些小行星，以及将小行星投向空间站。

首先创建原型小行星。小行星由两个对象构成：高层面的抽象对象，包含碰撞器及全部逻辑；一个“图形”对象，负责向玩家提供小行星的视觉表现。

(1) **创建对象**。创建一个新的空游戏对象，命名为 Asteroid。

(2) **向其添加小行星模型**。在 Models 文件夹中找到 Asteroid 模型，拖放到刚刚创建的 Asteroid 对象上，并将新的子对象命名为 Graphics。重置 Graphics 对象 Transform 组件的 Position，使其位于 (0, 0, 0)。

(3) **为 Asteroid 对象添加一个刚体和一个球体碰撞器**。不要添加到 Graphics 对象上。添加之后，关闭刚体的重力，并设球体碰撞器的半径为 2。

(4) **添加 `Asteroid` 脚本**。为 Asteroid 游戏对象添加一个新的 C# 脚本，命名为 Asteroid.cs，并添加下面的代码：

```
public class Asteroid : MonoBehaviour {

  //小行星的移动速度
  public float speed = 10.0f;

  void Start () {
    //设置刚体的速度
```

```
        GetComponent<Rigidbody>().velocity
          = transform.forward * speed;

        //为此小行星创建一个红色的指示器
        var indicator = IndicatorManager.instance
          .AddIndicator(gameObject, Color.red);

      }

    }
```

`Asteroid` 脚本很简单：当对象出现时，向对象的刚体应用一个“前向”力，使其开始向前移动。另外，告诉 Indicator Manager 在屏幕上为这个小行星添加一个新的指示器。



你会看到一个警告，指出 `indicator` 变量被写入，但没有被读取。这没有关系，在游戏中不会导致 bug。我们将在后面添加更多代码来使用 `indicator` 变量，到时候这个警告就会消失。

完成之后，Asteroid 的 Inspector 应该如图 12-1 所示。

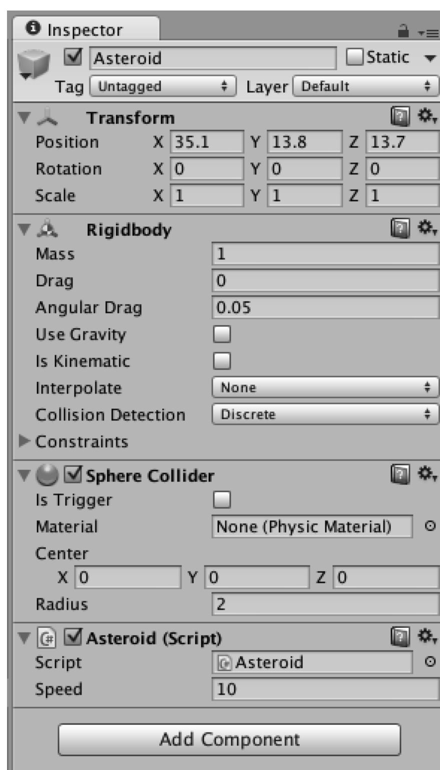


图 12-1：配置好的小行星

对象应该如图 12-2 所示。

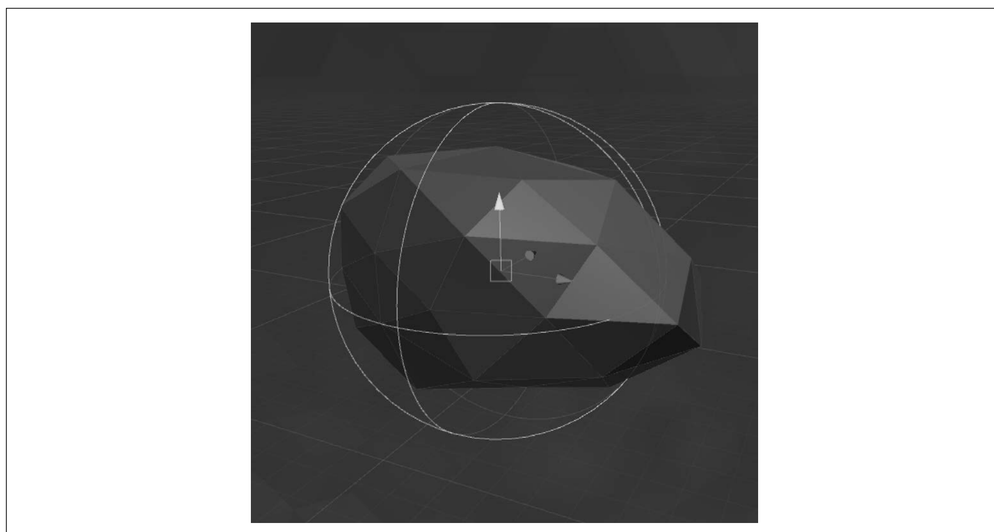


图 12-2：游戏中的小行星

(5) **测试小行星**。启动游戏，观察小行星。小行星应该向前移动，屏幕上将出现一个指示器。

Asteroid Spawner

设置好了小行星，接下来就要创建**小行星生成器**（asteroid spawner）了。小行星生成器是一个对象，它会定期创建新的小行星对象，并使它们瞄准目标。它将在一个不可见球面的随机点上创建小行星，并配置这些小行星，使它们的“前向”力对准游戏中的某个对象。另外，小行星生成器将使用 Unity 的 Gizmos 功能，可视化小行星所出现的空间范围。这是因为 Gizmos 功能允许在场景视图中显示额外的信息。

首先，把上一节创建的原型小行星转换为预设。然后，创建并设置 Asteroid Spawner。

(1) **将小行星转换为预设**。将 Asteroid 对象从 Hierarchy 窗格拖放到 Project 窗格。这将从该对象创建一个预设。然后，从场景中删除 Asteroid。

(2) **创建 Asteroid Spawner**。创建一个新的空游戏对象，命名为 Asteroid Spawner，设其位置为 (0, 0, 0)。

接下来，添加一个新的 C# 脚本，命名为 AsteroidSpawner.cs，并添加下面的代码：

```
public class AsteroidSpawner : MonoBehaviour {

  //生成区域的半径
  public float radius = 250.0f;

  //要生成的小行星
  public Rigidbody asteroidPrefab;

  //在生成每个小行星之间等待spawnRate±variance秒
```

```
public float spawnRate = 5.0f;
public float variance = 1.0f;

//将小行星对准此对象
public Transform target;

//如果为false，就禁用生成操作
public bool spawnAsteroids = false;

void Start () {
  //启动协程，立即创建小行星
  StartCoroutine(CreateAsteroids());
}

IEnumerator CreateAsteroids() {

  //无限循环
  while (true) {

    //计算下一个小行星的出现位置
    float nextSpawnTime
      = spawnRate + Random.Range(-variance, variance);

    //等待这么多秒
    yield return new WaitForSeconds(nextSpawnTime);

    //另外，等待物理系统更新
    yield return new WaitForFixedUpdate();

    //创建小行星
    CreateNewAsteroid();
  }

}

void CreateNewAsteroid() {

  //如果当前没有生成小行星，就返回
  if (spawnAsteroids == false) {
    return;
  }

  //在球体表面上随机选择一个点
  var asteroidPosition = Random.onUnitSphere * radius;

  //按对象的比例缩放
  asteroidPosition.Scale(transform.lossyScale);

  //使用小行星生成器的位置进行偏移
  asteroidPosition += transform.position;

  //创建新的小行星
  var newAsteroid = Instantiate(asteroidPrefab);

  //将其置于我们刚才计算出的位置
```

```
            newAsteroid.transform.position = asteroidPosition;

            //使其对准目标
            newAsteroid.transform.LookAt(target);
        }

        //选择生成器对象时由编辑器调用
        void OnDrawGizmosSelected() {

            //我们想要绘制黄色
            Gizmos.color = Color.yellow;

            //告诉Gizmos绘制器使用当前的位置和比例
            Gizmos.matrix = transform.localToWorldMatrix;

            //绘制一个球体，代表生成区域
            Gizmos.DrawWireSphere(Vector3.zero, radius);
        }

        public void DestroyAllAsteroids() {
            //移除游戏中的全部小行星
            foreach (var asteroid in
                FindObjectsOfType<Asteroid>()) {
                Destroy (asteroid.gameObject);
            }
        }
    }
```

`AsteroidSpawner` 脚本使用协程 `CreateAsteroids` 来连续创建新的小行星对象。它调用 `CreateNewAsteroid`，等待一会儿，然后重复这个过程。

另外，当选中小行星时，`OnDrawGizmosSelected` 方法会使其周围出现一个球体线框。这个球体代表小行星出现的位置：它们将出现在球形表面，并朝向目标移动。

(3) **压平 Asteroid Spawner**。将 Asteroid Spawner 的 Scale 设为 (1, 0.1, 1)。这将使大部分小行星出现在围绕其目标的一个圆圈上，而不是一个球面上（如图 12-3 所示）。

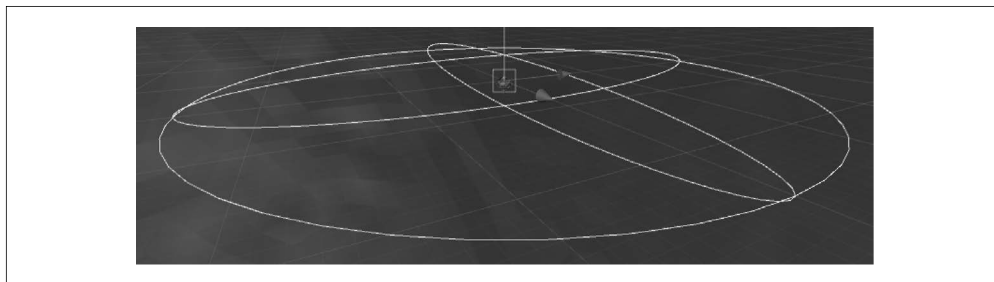


图 12-3：场景视图中的 Asteroid Spawner

(4) **配置** `AsteroidSpawner`。将刚才创建的 Asteroid 预设拖放到 Asteroid Prefab 框中，将 Space Station 对象拖放到 Target 框中。打开 Spawn Asteroids。

(5) **测试游戏**。Asteroid 将开始出现，并朝空间站移动！

12.2　造成伤害与受到伤害

现在，飞船就可以在空间站和朝空间站移动的小行星之间穿梭飞行，但是你发射的武器实际上还没什么作用。我们需要添加造成伤害和响应伤害的能力。

在这个游戏中，“伤害”意味着某些对象具有“生命值”。如果一个对象的生命值降低到 0，则从游戏中移除该对象。

有些对象能够造成伤害，有些对象能够受到伤害。还有些对象既能够造成伤害，又能够受到伤害，例如小行星，既会被激光束所伤害，又会伤害它们所击中的对象（例如空间站）。

为了实现这种效果，我们创建了两个脚本：`DamageTaking` 与 `DamageOnCollide`。

- `DamageTaking` 脚本维护自己所附加到的对象的剩余生命值，当生命值为 0 时，就从游戏中移除对象。`DamageTaking` 还提供了一个方法 `TakeDamage`，其他对象调用此方法，对此对象造成伤害。
- 当对象与其他对象碰撞时，或者进入触发器区域时，`DamageOnCollide` 脚本的代码会运行。如果碰撞到的对象具有 `DamageTaking` 组件，那么 `DamageOnCollide` 脚本会调用该对象的 `TakeDamage` 方法。

我们把 `DamageOnCollide` 脚本添加到 Shot 和 Asteroid，将 `DamageTaking` 脚本添加到 Space Station 和 Asteroid。

首先，我们让小行星能够受到伤害。

(1) **向小行星添加 `DamageTaking` 脚本**。在 Project 窗格中选择 Asteroid 预设，为其添加一个新的 C# 脚本，命名为 DamageTaking.cs，并在文件中添加下面的代码：

```
public class DamageTaking : MonoBehaviour {

  //此对象的生命值
  public int hitPoints = 10;

  //如果被摧毁，则在当前位置创建这样一个预设
  public GameObject destructionPrefab;

  //如果此对象被销毁，要结束游戏吗？
  public bool gameOverOnDestroyed = false;

  //其他对象（如Asteroid和Shot）调用此方法来受到伤害
  public void TakeDamage(int amount) {

    //报告：我们被击中
    Debug.Log(gameObject.name + " damaged!");

    //降低生命值
    hitPoints -= amount;

    //是否死亡
    if (hitPoints <= 0) {

      //记录下来
```

```
      Debug.Log(gameObject.name + " destroyed!");

      //从游戏中移除
      Destroy(gameObject);

      //我们有要使用的摧毁预设吗？
      if (destructionPrefab != null) {

        //在当前位置使用旋转值创建摧毁预设
        Instantiate(destructionPrefab,
          transform.position, transform.rotation);
      }

    }

  }

}
```

DamageTaking 脚本跟踪对象的生命值，并提供了一个方法，其他对象调用这个方法来对其造成伤害。如果生命值变成 0 或者更低，就销毁该对象。如果提供了摧毁预设（例如下一节将会添加的爆炸），还会创建一个摧毁预设。

(2) **配置小行星**。将小行星的 Hit Points 变量改为 1。这将使小行星很容易被摧毁。

接下来，我们使 Shot 对象对其击中的任意对象造成伤害。

(3) **为武器添加 DamageOnCollide 脚本**。选择 Shot 预设，为其添加一个新的 C# 脚本，命名为 DamageOnCollide.cs，并在文件中添加下面的代码：

```
public class DamageOnCollide : MonoBehaviour {

  //施加给所击中对象的伤害值
  public int damage = 1;

  //击中对象时，自己受到的伤害值
  public int damageToSelf = 5;

  void HitObject(GameObject theObject) {
    //如果可以，对击中的对象施加伤害
    var theirDamage =
      theObject.GetComponentInParent<DamageTaking>();
    if (theirDamage) {
      theirDamage.TakeDamage(damage);
    }

    //如果可以，对自己施加伤害
    var ourDamage =
      this.GetComponentInParent<DamageTaking>();
    if (ourDamage) {
      ourDamage.TakeDamage(damageToSelf);
    }
  }
```

```
    //有对象进入此触发器区域了吗？
    void OnTriggerEnter(Collider collider) {
      HitObject(collider.gameObject);
    }

    //有对象与我们发生碰撞了吗？
    void OnCollisionEnter(Collision collision) {
      HitObject(collision.gameObject);
    }
  }
```

DamageOnCollide 脚本也非常简单。如果检测到碰撞，或者对象与其触发器的碰撞器相交（飞船的情形），就调用 HitObject 方法，该方法检查碰撞到的对象是否有 DamageTaking 组件。如果有，就调用该组件的 TakeDamage 方法。另外，我们对**当前**的对象执行相同的操作。这么做是因为，如果小行星击中空间站，我们想要销毁小行星，并让空间站受到一些伤害。

(4) **测试游戏**。四处飞行，并射击小行星。当武器击中一个小行星时，小行星将会消失。

接下来，我们使空间站可被摧毁。

(5) **向空间站添加** DamageTaking。选择空间站，添加一个 DamageTaking 脚本组件。
打开 Game Over On Destruction。现在这个选项还没有什么作用，但是到后面摧毁空间站时，我们将用它来使游戏结束。
完成之后，Space Station 的 Inspector 应该如图 12-4 所示。

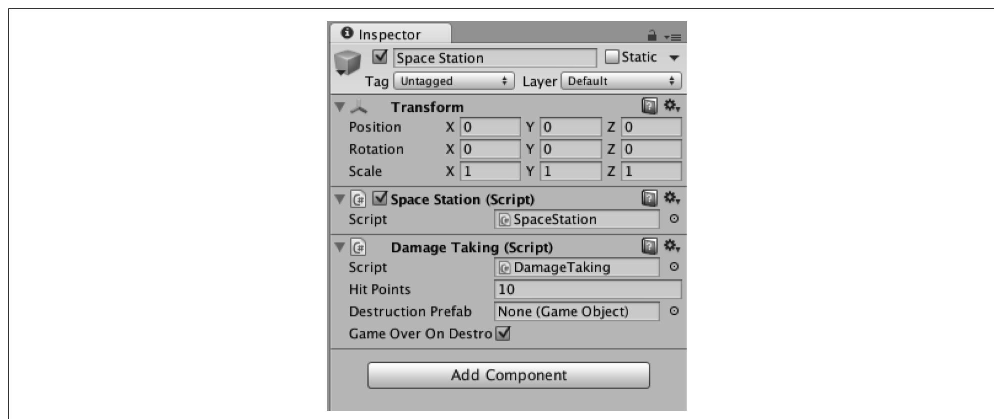


图 12-4：向空间站添加 DamageTaking 脚本

爆炸

当一个小行星被摧毁时，它只是简单地消失掉。这种效果并不是特别让人满意，更好的方法是让小行星先爆炸，然后消失。

使用粒子效果是创建爆炸最好的方式之一。想要实现看起来很自然的随机效果时，使用粒子效果就很合适。它可以用来表现烟雾、火焰、风以及爆炸。

这个游戏中的爆炸由**两种**粒子效果构成：第一种粒子效果将创建一开始的闪光；第二种粒子效果将留下一些逐渐消失的烟尘。

使用粒子效果时，准备好资源是很重要的。你尤其需要决定的是，让粒子效果使用自定义材质，还是使用默认的粒子材质。默认材质是一种模糊的圆形，能够很好地表现许多东西，但是如果要为自己的效果添加更多细节，就需要创建自己的材质。

闪光效果可以使用默认的粒子材质，但是烟尘效果需要创建一个自定义材质。尽管通过使用很多微小的实例，也**可以**用默认材质来创建烟尘，但是如果使用烟尘的图片作为起点，就可以事半功倍。

(1) **创建 Dust 材质**。打开 Asset 菜单，选择 Create → Material。将新材质命名为 Dust。

(2) **配置材质**。选择材质，将其着色器改为 Particles/Additive。

接下来，将 Dust 纹理拖放到 Particle Texture 框中。

单击 Tint Color 框并选择颜色，设为半透明的深灰色。如果你更喜欢输入具体数值，则输入 (70, 70, 70, 190)，如图 12-5 所示。

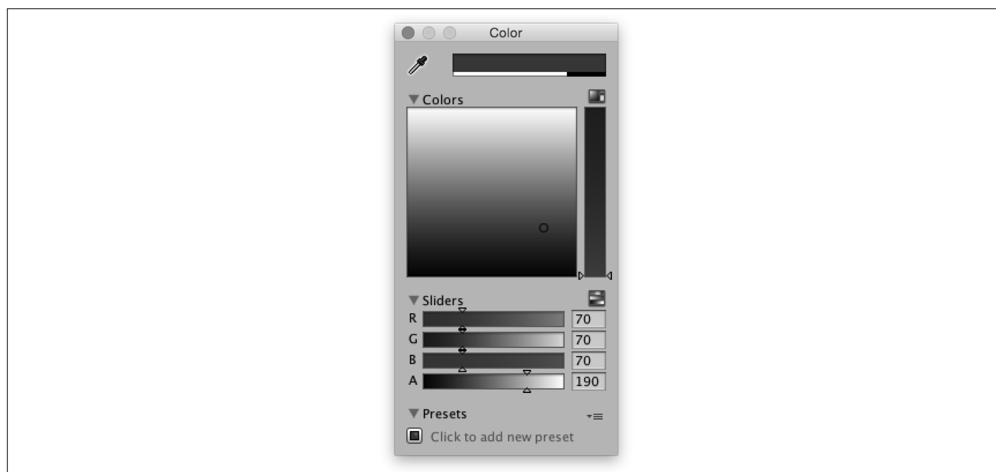


图 12-5：Dust 材质的着色

最后，将 Soft Particles Factor 设为 0.8。

完成之后，材质的 Inspector 应该如图 12-6 所示。

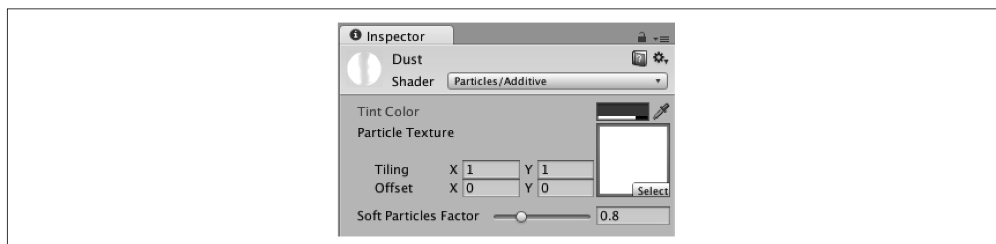


图 12-6：Dust 材质

现在就可以创建粒子系统。首先，我们为爆炸创建一个空的容器对象，然后创建并设置两个粒子系统。

(1) **创建 Explosion 对象**。创建一个新的空对象，命名为 Explosion。

(2) **创建 Fireball 对象**。再创建一个空对象，命名为 Fireball，设为 Explosion 对象的子对象。

(3) **为 Fireball 添加并配置粒子效果**。选择 Fireball，添加一个新的 Particle Effect 组件。按照图 12-7 所示设置粒子效果。

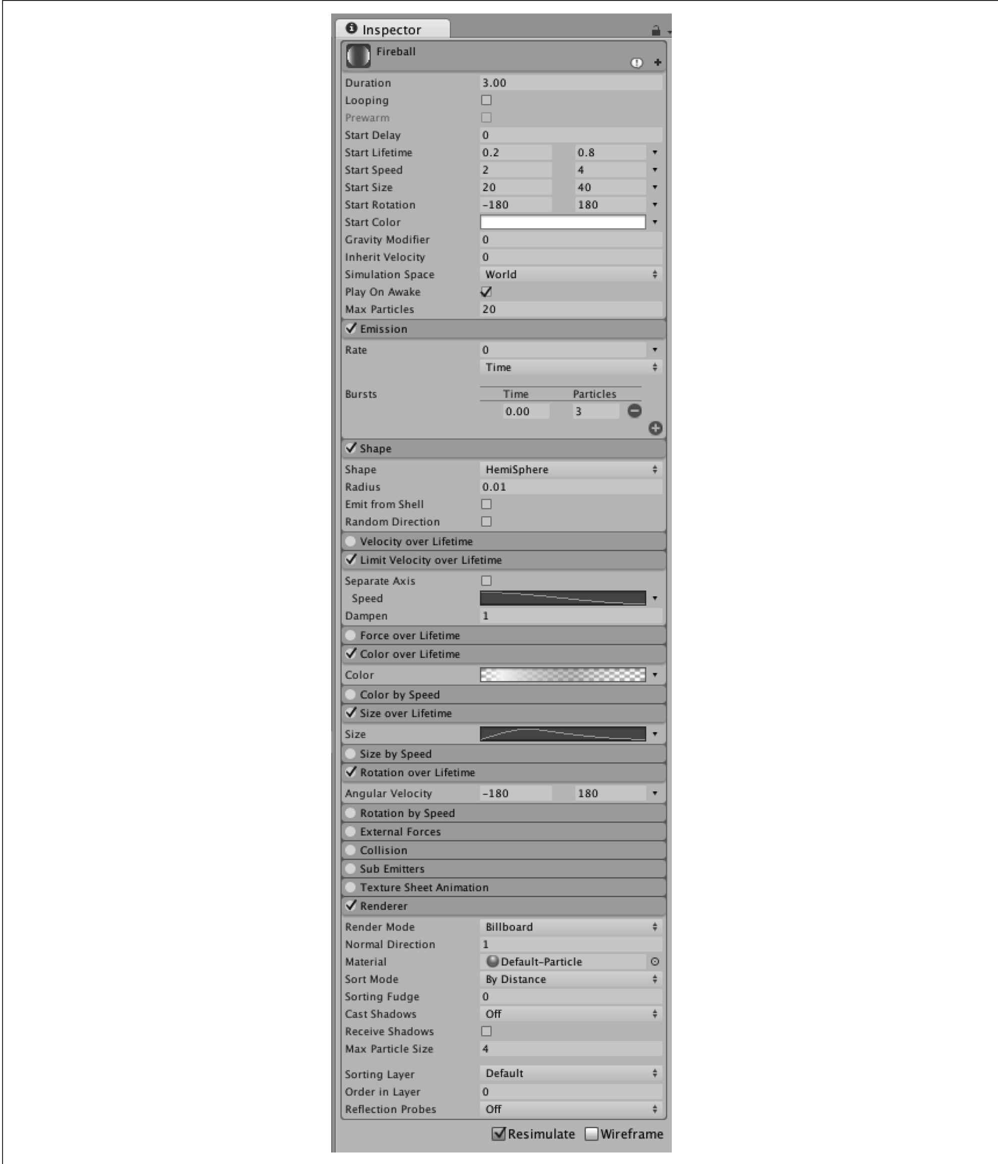


图 12-7：Fireball 的粒子效果的 Inspector



大部分参数都是可以直接输入的数值，但是有几个参数需要解释一下。

- Color over Lifetime 渐变如图 12-8 所示。

渐变的 alpha 值为：

- 0% 位置为 0
- 12% 位置为 255
- 100% 位置为 0

颜色值为：

- 0% 位置为白色
- 12% 位置为浅棕褐色
- 15% 位置为深棕褐色
- 100% 位置为白色

Size over Lifetime 一开始为 0，在 35% 的位置为 3，在结束时又变为 0（如图 12-9 所示）。

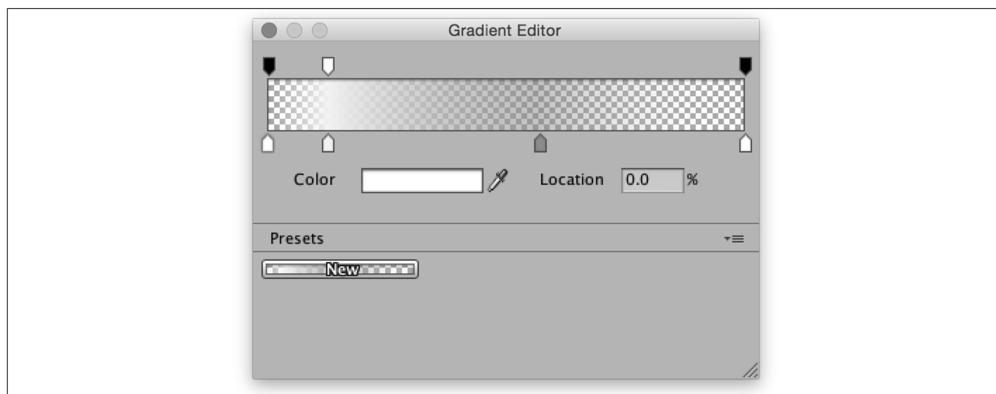


图 12-8：爆炸火球的 Color over Lifetime 渐变

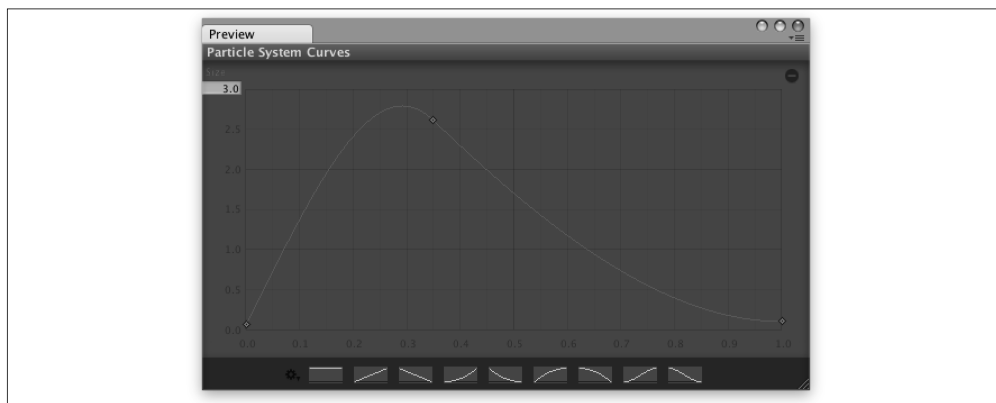


图 12-9：爆炸火球的 Size over Lifetime 曲线（另见彩插）

Fireball 对象创建了爆炸开始时的短暂闪光。接下来要添加的粒子效果是 Dust 效果。

(1) **创建 Dust 对象**。创建一个空游戏对象，命名为 Dust，设为 Explosion 对象的子对象。

(2) **添加并配置粒子系统**。添加一个新的 Particle System 组件，按照图 12-10 所示进行设置。

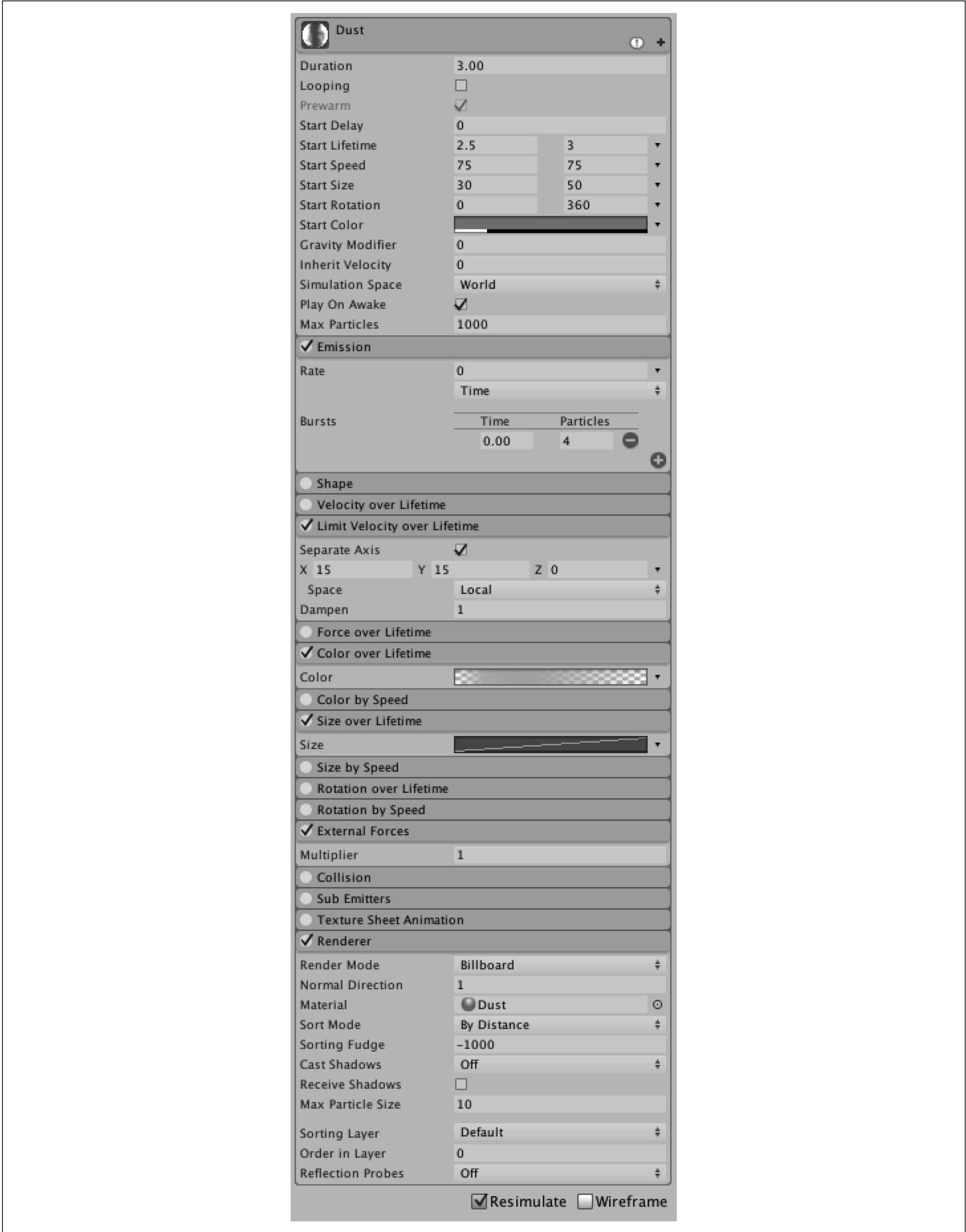


图 12-10：Dust 粒子的 Inspector



有几个地方不能直接从图 12-10 复制，说明如下。

- Renderer 使用的 Material 就是刚才创建的 Dust 材质，需要把该材质拖放到 Material 框中。
- 初始颜色为 RGBA 值 [130, 130, 120, 45]。单击 Start Color 变量，然后输入这些值。
- Size over Lifetime 是一条直线，从 0% 变化到 100%。
- Color over Lifetime 如图 12-11 所示。颜色是棕褐色不变，Alpha 值在 0% 位置为 0，到 14% 位置变为 255，到 100% 位置变为 0。

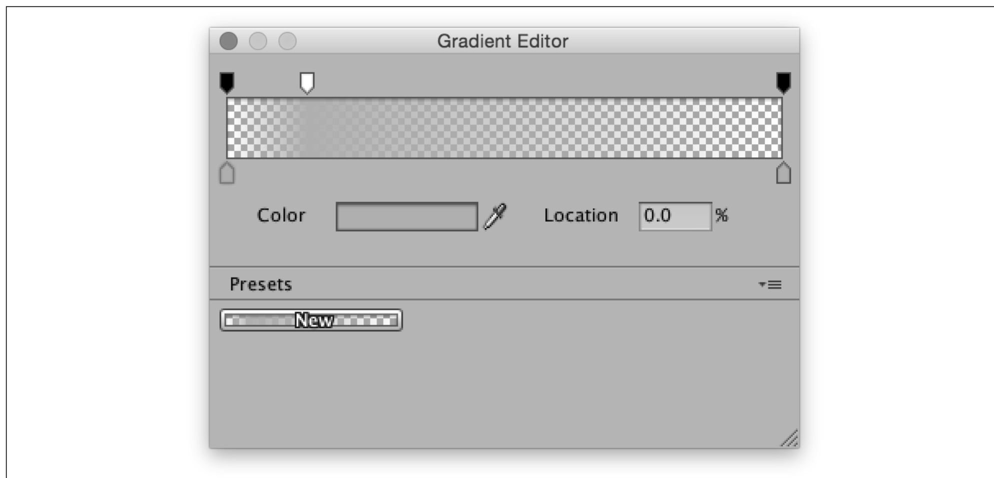


图 12-11：爆炸烟尘粒子的 Color over Lifetime 设置

现在就完成了对粒子系统的设置，可以为小行星使用这个爆炸效果了。

(1) **将 Explosion 对象转换为预设**。将 Explosion 对象拖放到 Project 窗格，然后将其从场景中移除。

(2) **使小行星在被摧毁时使用爆炸效果**。选择 Asteroid 预设，将 Explosion 拖入 Destruction Prefab 框中。

(3) **进行测试**。击落小行星时，它们将会爆炸！

12.3 小结

现在已经创建了小行星和伤害模型，游戏就很接近于完整了。第 13 章我们将开始优化游戏，使其成为一个使用体验更加丰富且美好的游戏。

第13章

音效、菜单、死亡及爆炸

太空射击游戏的核心游戏玩法已基本完成，但是游戏还没有达到完善的程度。为了在Unity以外也能够玩游戏，需要添加菜单和其他控制，让玩家能够在游戏中导航。最后，我们将用更高保真度的3D模型和材质替换临时使用的美术作品，使游戏更加精美。

13.1 菜单

目前的游戏玩法只能使用编辑器的Play按钮。启动游戏后，游戏就立即开始，而如果空间站被摧毁，就必须停止游戏，然后再次启动。

为了给玩家提供更多的游戏情境，我们需要添加菜单，尤其要添加一个非常重要的按钮——New Game。如果空间站被摧毁，我们需要提供一种让玩家再次开始游戏的方法。

为游戏添加菜单结构，能够大大提升游戏的完整度。作为菜单的一部分，我们将添加如下4个组件。

Main Menu
　此画面显示游戏名称，以及New Game按钮。

Paused 画面
　此画面显示文本Paused，并包含一个恢复游戏的按钮。

Game Over 画面
　此画面显示Game Over，以及New Game按钮。

In-Game UI
　此画面包含摇杆、指示器、Fire按钮，以及玩家在玩游戏时实际看到的所有东西。

这些UI分组是互斥的，一次只能显示一个。游戏开始时将显示Main Menu，单击New

Game 按钮时，Main Menu 将被替换为 In-Game UI，实际的游戏也会开始。



Unity 的 UI 系统允许使用计算机的鼠标或触摸板测试菜单。但是在开发游戏时，仍然应该测试菜单在真正的触摸屏上的感觉，例如使用 Unity Remote 应用帮助测试（参见 5.1.1 节）。

这个过程的第一步是把 In-Game 组件放到一个对象中，以便能够同时管理。

(1) **创建 In-Game UI 容器**。选择 Canvas 对象，创建一个新的空子对象，命名为 In-Game UI。

(2) **配置容器**。使 In-Game UI 的锚点水平和垂直拉伸，并将 Left、Top、Bottom 和 Right 边距设为 0。这将使其填满整个画布。

接下来，我们把全部已经存在的 UI 元素放到整个容器中。

(3) **组合游戏的 UI**。选择画布的每个子对象（但 In-Game UI 容器**除外**），把它们移动到 In-Game UI 中。

现在开始构建其他菜单。在开始之前，先关闭 In-Game UI，以免影响到将要构建的其他 UI 内容。

(4) **禁用 In-Game UI**。选择 In-Game UI 对象，单击 Inspector 左上角的复选框来禁用它。完成之后，Inspector 应该如图 13-1 所示。

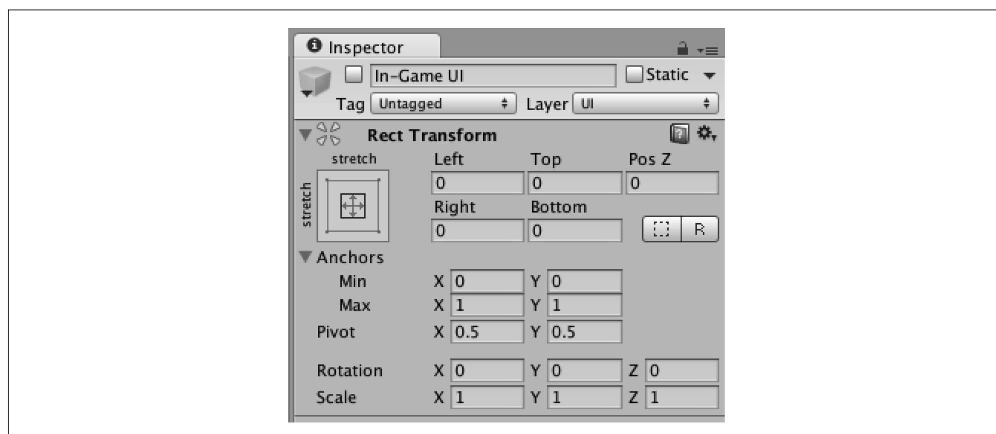


图 13-1：禁用后的 In-Game UI；另外要注意该对象的大小和位置，它被设为填满整个画布，并且不带边框

13.1.1　主菜单

Main Menu（主菜单）的内容非常简单，只包含一个显示游戏名称（Rockfall）的文本标签，以及一个创建新游戏的按钮。

与 In-Game UI 类似，Main Menu 由一个空的容器对象构成，所有属于主菜单的 UI 组件都将是这个容器对象的子对象。

(1) **创建 Main Menu 容器**。创建一个新的空游戏对象，使其成为 Canvas 的子对象。将其命名为 Main Menu。

将此对象设为水平和垂直拉伸，以填充整个画布。将全部边距值设为 0。

(2) **创建名称标签**。打开 GameObject 菜单，选择 UI → Text，创建一个新的 Text 对象。使其成为 Main Menu 的子对象，命名为 Title。

将整个新 Text 对象的锚点设为 Center Top。将 *x* 位置值设为 0，*y* 位置值设为 −100。将高度设为 120，宽度设为 1024。

接下来需要设置文本。将文本的颜色设为十六进制颜色值 #FFE99A（稍微带点黄色），对齐方式设为居中，将文本自身设为 Rockfall。另外，启用 Best Fit 设置，这将使文本自动调整大小，以适应 Text 对象的边界。最后，将 At Night 字体拖入 Font 框中。

(3) **创建 Button**。创建一个新的 Button 对象，命名为 New Game，设为 Main Menu 的子对象。

将按钮的锚点设为 Center Top，*x* 和 *y* 位置值设为 [0, −300]。将其宽度设为 330，高度设为 80。

将按钮的 Source Image 设为 `Button` 精灵，将其 Image Type 设为 Sliced。

选择 Text 子对象，将其文本值改为 New Game。将字体设为 CRYSTAL-Regular，字号设为 28，颜色设为 3DFFD0FF。

完成之后，主菜单应该如图 13-2 所示。



图 13-2：Main Menu

在继续设置之前，先禁用 Main Menu 容器。

13.1.2　Paused画面

Paused 画面显示文本 Paused，以及一个用来恢复游戏的按钮。在构建这个画面时，执行的步骤与 Main Menu 相同，但是需要做如下的修改。

- 将容器对象命名为 Paused。
- 将 Title 对象的文本设为 Paused。
- 将 Button 对象命名为 Unpause Button。
- 将 Button 对象的文本设为 Resume。

完成之后，Pause 菜单应该如图 13-3 所示。



图 13-3：Pause 菜单

在构建最后一个菜单（Game Over 画面）之前，先禁用 Paused 容器。

13.1.3　Game Over画面

Game Over 画面显示文本 Game Over，以及一个开始新一轮游戏的按钮。当空间站被摧毁时，游戏结束，此时将显示 Game Over 画面。

同样，按照构建 Main Menu 和 Paused 画面的步骤来构建 Game Over 画面，但是有以下几处改动。

- 将容器对象命名为 Game Over。
- 将 Title 对象的文本设为 Game Over。
- 将 Button 对象命名为 New Game Button。
- 将 Button 对象的文本设为 New Game。

完成之后，Game Over 画面应该如图 13-4 所示。



图 13-4：Game Over 菜单



这 3 个新菜单基本上是相同的，所以你可能在想，为什么要做 3 次同样的工作？原因在于，以后你会想要定制这些菜单，所以现在就把它们拆分开，有助于节省以后的工作量。

我们还需要向游戏添加最后一个 UI 组件，作为一种暂停游戏的方法。

13.1.4　添加Pause按钮

Pause 按钮出现在 In-Game UI 的右上角，用于告诉游戏，用户想要暂停一会儿。

构建 Pause 按钮，首先需要创建一个新的 Button 对象，使其成为 In-Game UI 容器对象的子对象，并命名为 Pause Button。

将 Pause 按钮的锚点设为 Top Right，将其 x 和 y 位置值设为 [−50, −30]。将宽度设为 80，高度设为 70。

将其 Image 组件的 Source Image 设为 `Button` 精灵。

将 Text 子对象的文本设为 Pause。将其字体设为 CRYSTAL-Regular，字号设为 28。将其颜色设为 #3DFFD0FF。

祝贺你！ UI 现在已经完成了。但是，我们设置的按钮还不能正常工作。为了解决这个问题，需要添加一个 Game Manager 来进行协调。

13.2　Game Manager和死亡

与 Input Manager 和 Indicator Manager 类似，Game Manager 是一个单例对象。Game Manager 有如下两个主要任务：

- 管理游戏的状态和菜单
- 生成飞船和空间站

游戏启动时，将处在未开始的状态。飞船和空间站不在场景中，小行星生成器也没有创建小行星。另外，Game Manager 将显示 Main Menu 容器对象，而隐藏所有其他菜单。

当用户触摸 New Game 按钮时，将会显示 In-Game UI，创建飞船和空间站，并告诉小行星生成器开始创建小行星。另外，Game Manager 将设置游戏中的一些重要元素：告诉 `Camera Follow` 脚本跟随新的 Ship 对象，并告诉 Asteroid Spawner 将其小行星对准 Space Station。

最后，Game Manager 将处理 Game Over 状态。你可能记得前面提到过，`DamageTaking` 脚本有一个复选框，叫作 Game Over On Destroyed。我们将进行相应设置，如果勾选该复选框，那么每当脚本所附加到的对象被摧毁时，Game Manager 就结束游戏。结束游戏很简单，只需要关闭小行星生成器，并销毁当前的飞船（如果空间站还在，还需要销毁空间站）。

在开始构建 Game Manager 之前，需要能够创建 Ship 和 Space Station 的多个副本。这需要把这两个对象转换为预设，同时定义它们出现的位置。

将 Ship 和 Space Station 转换为预设。将 Ship 拖放到 Project 窗格来创建预设，然后从场景中移除 Ship。对 Space Station 重复这个过程。

13.2.1 起始点

现在将创建两个标记对象作为指示器，当新游戏启动时，指定创建 Ship 和 Space Station 的位置。玩家看不到这些指示器，但是我们将使你能够在编辑器内看到它们。

(1) **创建 Ship 位置标记**。创建一个新的空游戏对象，命名为 Ship Start Point。
单击 Inspector 左上角的图标，选择红色的标签（如图 13-5 所示）。该对象现在将出现在场景视图中，尽管对玩家是不可见的。
将标记放到你希望飞船出现的位置。

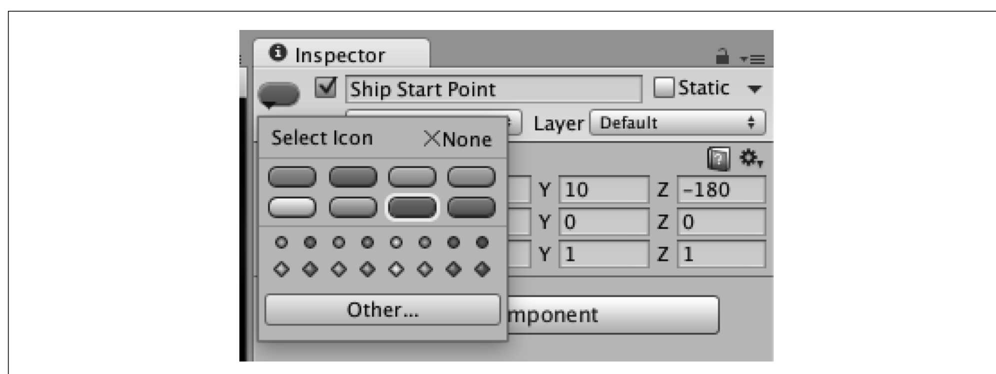


图 13-5：为飞船的起始点选择一个标签

(2) **创建 Space Station 位置标记**。重复上面的步骤，但是这一次创建一个名为 Station Start Point 的对象。将其放到你希望空间站出现的位置。

完成这些设置之后，我们就能够创建和设置 Game Manager 了。

13.2.2 创建Game Manager

Game Manager 在很大程度上作为一个中心点，用来存储游戏的关键信息（例如对当前 Ship 和 Space Station 的引用），以及修改重要游戏对象的状态（当单击了某个按钮，或者 `DamageTaking` 脚本指出游戏应该结束时）。

为了设置 Game Manager，创建一个新的空游戏对象，命名为 Game Manager。为其添加一个 C# 脚本，命名为 GameManager.cs，在文件中添加下面的代码：

```
public class GameManager : Singleton<GameManager> {

  //为飞船使用的预设，飞船的起始位置，以及当前的飞船对象
  public Gameobject shipPrefab;
  public Transform shipStartPosition;
  public GameObject currentShip {get; private set;}
```

```
//为空间站使用的预设，空间站的起始位置，以及当前的空间站对象
public GameObject spaceStationPrefab;
public Transform spaceStationStartPosition;
public GameObject currentSpaceStation {get; private set;}

//主摄像机上的跟随脚本
public SmoothFollow cameraFollow;

//各个UI部分的容器
public GameObject inGameUI;
public GameObject pausedUI;
public GameObject gameOverUI;
public GameObject mainMenuUI;

//当前在玩游戏吗？
public bool gameIsPlaying {get; private set;}

//游戏的小行星生成器
public AsteroidSpawner asteroidSpawner;

//跟踪游戏是否暂停
public bool paused;

//当游戏启动时显示主菜单
void Start() {
  ShowMainMenu();
}

//显示一个UI容器，并隐藏其他所有UI容器
void ShowUI(GameObject newUI) {

  //创建所有UI容器的列表
  GameObject[] allUI
    = {inGameUI, pausedUI, gameOverUI, mainMenuUI};

  //隐藏全部UI容器
  foreach (GameObject UIToHide in allUI) {
    UIToHide.SetActive(false);
  }

  //然后显示提供的UI容器
  newUI.SetActive(true);
}

public void ShowMainMenu() {
  ShowUI(mainMenuUI);

  //游戏启动时，我们还没有玩游戏
  gameIsPlaying = false;

  //也不生成小行星
  asteroidSpawner.spawnAsteroids = false;
}

//由被触摸的New Game按钮调用
```

```
public void StartGame() {
  //显示In-Game UI
  ShowUI(inGameUI);

  //我们现在在玩游戏
  gameIsPlaying = true;

  //如果刚好有飞船，就销毁该飞船
  if (currentShip != null) {
    Destroy(currentShip);
  }

  //对空间站执行类似处理
  if (currentSpaceStation != null) {
    Destroy(currentSpaceStation);
  }

  //创建一个新飞船，置于起始位置
  currentShip = Instantiate(shipPrefab);
  currentShip.transform.position
    = shipStartPosition.position;
  currentShip.transform.rotation
    = shipStartPosition.rotation;

  //对空间站执行类似处理
  currentSpaceStation = Instantiate(spaceStationPrefab);

  currentSpaceStation.transform.position =
    spaceStationStartPosition.position;

  currentSpaceStation.transform.rotation =
    spaceStationStartPosition.rotation;

  //使跟随脚本跟踪新飞船
  cameraFollow.target = currentShip.transform;

  //开始生成小行星
  asteroidSpawner.spawnAsteroids = true;

  //使生成器对准新的空间站
  asteroidSpawner.target = currentSpaceStation.transform;

}

//由被摧毁时结束游戏的对象调用
public void GameOver() {
  //显示Game Over UI
  ShowUI(gameOverUI);

  //我们不再玩游戏
  gameIsPlaying = false;

  //销毁飞船和空间站
  if (currentShip != null)
    Destroy (currentShip);
```

```
    if (currentSpaceStation != null)
      Destroy (currentSpaceStation);

    //停止生成小行星
    asteroidSpawner.spawnAsteroids = false;

    //移除游戏中所有残存的小行星
    asteroidSpawner.DestroyAllAsteroids();
  }

  //当触摸Pause或Resume按钮时调用
  public void SetPaused(bool paused) {

    //在游戏内UI和暂停UI之间切换
    inGameUI.SetActive(!paused);
    pausedUI.SetActive(paused);

    //如果已暂停……
    if (paused) {
      //停止时间
      Time.timeScale = 0.0f;
    } else {
      //恢复时间
      Time.timeScale = 1.0f;
    }
  }

}
```

Game Manager 脚本的代码很多，但是很简单。它有两个主要的功能：管理菜单和 In-Game UI 的显示；在游戏开始和结束时，创建和销毁空间站及飞船。

下面我们一步步介绍 Game Manager 脚本的功能。

1. 初始设置

当 Game Manager 在场景中第一次出现时，即游戏开始时，将调用 `Start` 方法。这个方法唯一的作用是调用 `ShowMainMenu` 来显示主菜单。

```
//当游戏启动时显示主菜单
void Start() {
  ShowMainMenu();
}
```

为了显示 UI，我们使用 `ShowUI` 方法来显示期望的 UI 对象，以及移除其他所有 UI 对象。具体来说，该方法隐藏所有 UI 对象，然后显示期望的 UI 元素，来实现此功能：

```
//显示一个UI容器，并隐藏其他所有UI容器
void ShowUI(GameObject newUI) {

  //创建所有UI容器的列表
  GameObject[] allUI
    = {inGameUI, pausedUI, gameOverUI, mainMenuUI};
```

```
    //隐藏全部UI容器
    foreach (GameObject UIToHide in allUI) {
      UIToHide.SetActive(false);
    }

    //然后显示提供的UI容器
    newUI.SetActive(true);
  }
```

实现了此方法后，就可以实现 ShowMainMenu 了。这个方法显示主菜单 UI（通过调用 ShowUI 方法），并指出游戏目前还没有开始，小行星生成器不应该生成小行星：

```
  public void ShowMainMenu() {
    ShowUI(mainMenuUI);

    //游戏启动时，我们还没有玩游戏
    gameIsPlaying = false;

    //也不生成小行星
    asteroidSpawner.spawnAsteroids = false;
  }
```

2. 开始游戏

触摸 New Game 按钮时，将调用 StartGame 方法。该方法显示 In-Game UI（这会隐藏其他 UI），并通过移除已有的以及创建新的飞船和空间站来设置游戏场景。它还使摄像机开始跟随新创建的飞船，并告诉小行星生成器开始朝着新创建的空间站投掷小行星：

```
  //由被触摸的New Game按钮调用
  public void StartGame() {
    //显示游戏内UI
    ShowUI(inGameUI);

    //我们现在在玩游戏
    gameIsPlaying = true;

    //如果刚好有飞船，就销毁该飞船
    if (currentShip != null) {
      Destroy(currentShip);
    }

    //对空间站执行类似处理
    if (currentSpaceStation != null) {
      Destroy(currentSpaceStation);
    }

    //创建一个新飞船，置于起始位置
    currentShip = Instantiate(shipPrefab);
    currentShip.transform.position
      = shipStartPosition.position;
    currentShip.transform.rotation
      = shipStartPosition.rotation;

    //对空间站执行类似处理
    currentSpaceStation = Instantiate(spaceStationPrefab);
```

```
    currentSpaceStation.transform.position =
      spaceStationStartPosition.position;

    currentSpaceStation.transform.rotation =
      spaceStationStartPosition.rotation;

    //使跟随脚本跟踪新飞船
    cameraFollow.target = currentShip.transform;

    //开始生成小行星
    asteroidSpawner.spawnAsteroids = true;

    //使生成器对准新的空间站
    asteroidSpawner.target = currentSpaceStation.transform;

  }
```

3. 结束游戏

特定的对象会调用 `GameOver` 方法，当它们被摧毁时，就会结束游戏。该方法显示 Game Over UI，停止游戏，并销毁当前的飞船和空间站。另外，它还会停止生成小行星，并移除游戏中全部剩余的小行星。本质上来说，我们返回了游戏的起始状态：

```
  //由被摧毁时结束游戏的对象调用
  public void GameOver() {
    //显示Game Over UI
    ShowUI(gameOverUI);

    //我们不再玩游戏
    gameIsPlaying = false;

    //销毁飞船和空间站
    if (currentShip != null)
      Destroy (currentShip);

    if (currentSpaceStation != null)
      Destroy (currentSpaceStation);

    //停止生成小行星
    asteroidSpawner.spawnAsteroids = false;

    //移除游戏中所有残存的小行星
    asteroidSpawner.DestroyAllAsteroids();
  }
```

4. 暂停游戏

当触摸 Pause 按钮或 Resume 按钮时，`SetPaused` 方法将被调用。该方法负责显示暂停 UI，以及停止或恢复时间流。

```
  //当触摸Pause或Resume按钮时调用
  public void SetPaused(bool paused) {
```

```
    //在In-Game UI和暂停UI之间切换
    inGameUI.SetActive(!paused);
    pausedUI.SetActive(paused);

    //如果已暂停……
    if (paused) {
      //停止时间
      Time.timeScale = 0.0f;
    } else {
      //恢复时间
      Time.timeScale = 1.0f;
    }
  }
```

13.2.3　设置场景

编写好代码后，现在就可以在场景中设置 Game Manager 了。配置 Game Manager 就是把场景中的对象与脚本中的变量连接起来。

- Ship Prefab 应该是我们刚才创建的飞船预设。
- Ship 的开始位置应该是场景中的飞船起始位置。
- Station Prefab 应该是我们刚才创建的空间站预设。
- Station 的起始位置应该是场景中的空间站起始位置。
- Camera Follow 应该是场景中的 Main Camera。
- In-Game UI、Main Menu UI、Paused UI 和 Game Over UI 应该是它们在场景中对应的 UI。
- Asteroid Spawner 应该是场景中的 Asteroid Spawner 对象。
- 现在暂不处理 Warning UI，这是下一节的内容。

完成之后，Game Manager 的 Inspector 应该如图 13-6 所示。

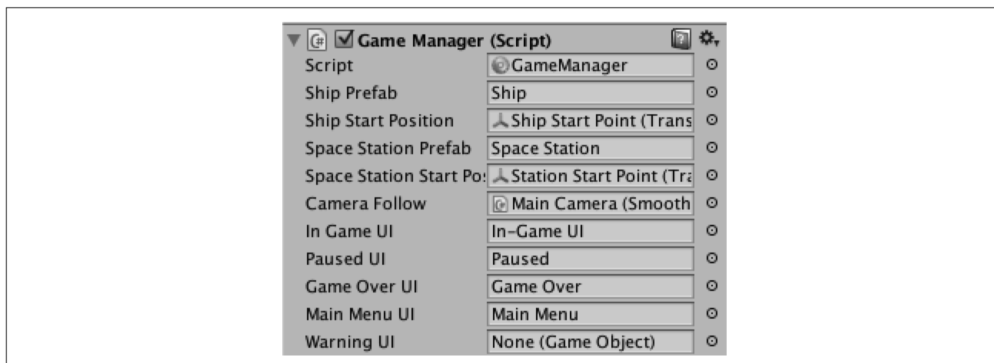


图 13-6：Game Manager 的 Inspector

设置好 Game Manager 之后，我们需要把 Game UI 中的各个按钮连接到 Game Manager。

(1) **连接 Pause 按钮**。在 In-Game UI 中选择 Pause 按钮，然后单击 Clicked 事件底部的 + 按钮。将 Game Manager 拖放到出现的框中，并将函数改为 GameManager → SetPaused。

此时将显示一个复选框，选中它。这样会调用 Game Manager 的 `SetPaused` 方法，并传入布尔值 `true`。

(2) **连接 Unpause 按钮**。在 Paused 菜单中选择 UnPause 按钮。执行与 Pause 按钮相同的步骤，但是要做一个修改：关闭复选框。这样，当按钮调用 `SetPaused` 方法时，将传入布尔值 `false`。

(3) **连接 New Game 按钮**。在 Main Menu 中选择 New Game 按钮，然后单击 Clicked 事件底部的 + 按钮。将 Game Manager 拖放到出现的框中，并将函数改为 GameManager → StartGame。接下来，为 Game Over 画面中的 New Game 按钮执行相同的步骤。

现在按钮就快设置好了。但在完成设置之前，我们还需要做一些小工作，以便能够获得完整的游戏体验。

首先，我们需要让空间站被摧毁时结束游戏。Space Station 已经附加了 `DamageTaking` 脚本。我们需要让这个脚本调用 Game Manager 的 `GameOver` 函数。

(4) **在 DamageTaking.cs 中添加对 `GameOver` 的调用**。打开该文件，添加下面的代码：

```
public class DamageTaking : MonoBehaviour {

  //此对象的生命值
  public int hitPoints = 10;

  //如果被摧毁，则在当前位置创建这样一个预设
  public GameObject destructionPrefab;

  //如果此对象被销毁，应该结束游戏吗？
  public bool gameOverOnDestroyed = false;

  //其他对象（如Asteroid和Shot）调用此方法来受到伤害
  public void TakeDamage(int amount) {

    //报告称我们被击中
    Debug.Log(gameObject.name + " damaged!");

    //减小我们的生命值
    hitPoints -= amount;

    //死亡了吗？
    if (hitPoints <= 0) {

      //记录下来
      Debug.Log(gameObject.name + " destroyed!");

      //把我们从游戏中移除
      Destroy(gameObject);

      //我们有要使用的摧毁预设吗？
      if (destructionPrefab != null) {

          //在当前位置使用旋转值创建摧毁预设
          Instantiate(destructionPrefab,
                transform.position, transform.rotation);
```

```
      }

>     //如果现在应该结束游戏，就调用GameManager的GameOver方法
>     if (gameOverOnDestroyed == true) {
>       GameManager.instance.GameOver();
>     }
    }

  }

}
```

这段代码使对象检查 `gameOverOnDestroyed` 变量是否设置为 `true`。如果是，就调用 Game Manager 的 `GameOver` 方法，使游戏结束。

我们还需要在发生撞击时，让小行星造成伤害。为此，我们将向小行星添加 `DamageOnCollide` 脚本。

为了使小行星应用伤害，选择 Asteroid 预设，然后添加一个 `DamageOnCollide` 组件。

接下来，小行星应该显示自己与空间站的距离。这有助于玩家确定哪个小行星是最应该注意的。为此，我们将修改 `Asteroid` 脚本，向 Game Manager 查询当前的空间站，然后将查找到的空间站赋值给小行星指示器的 `showDistanceTo` 变量。

为了使小行星显示距离标签，打开 Asteroid.cs，将下面的代码添加到 `Start` 函数中：

```
public class Asteroid : MonoBehaviour {

  //小行星的移动速度
  public float speed = 10.0f;

  void Start () {
    //设置刚体的速度
    GetComponent<Rigidbody>().velocity
      = transform.forward * speed;

    //为此小行星创建一个红色的指示器
    var indicator =
      IndicatorManager.instance.AddIndicator(
        gameObject, Color.red);

>     //跟踪从此对象到GameManager管理的当前空间站的距离
>     indicator.showDistanceTo =
>       GameManager.instance.currentSpaceStation
>         .transform;
  }

}
```

这段代码设置指示器，显示小行星与当前空间站的距离，帮助玩家判断哪些是最接近空间站的小行星。

现在我们就完成了对游戏的设置！

试玩游戏。你现在可以四处飞行并射击小行星。如果空间站被太多小行星击中，那么它将被摧毁。你自己也可以射击并摧毁空间站，当它被摧毁时，游戏就会结束！

13.3 边界

我们还需要添加最后一个游戏玩法核心元素：如果玩家飞离空间站太远，那么对其发出警告。如果玩家飞得太远，我们将在屏幕四周显示一个红色警告边框。如果玩家继续远离，游戏就会结束。

13.3.1 创建UI

首先，我们为警告创建 UI。

(1) **添加 Warning 精灵**。选择 Warning 纹理，将纹理的类型改为 Sprite/UI。

我们需要**切割**这个精灵，使其能够在整个屏幕内进行拉伸，而不会扭曲边角的形状。

(2) **切割精灵**。单击 Sprite Editor 按钮，精灵将在一个新窗口中显示。在该窗口右下角的面板中，将各边的边框设为 127。这使得边角不会被拉伸，如图 13-7 所示。

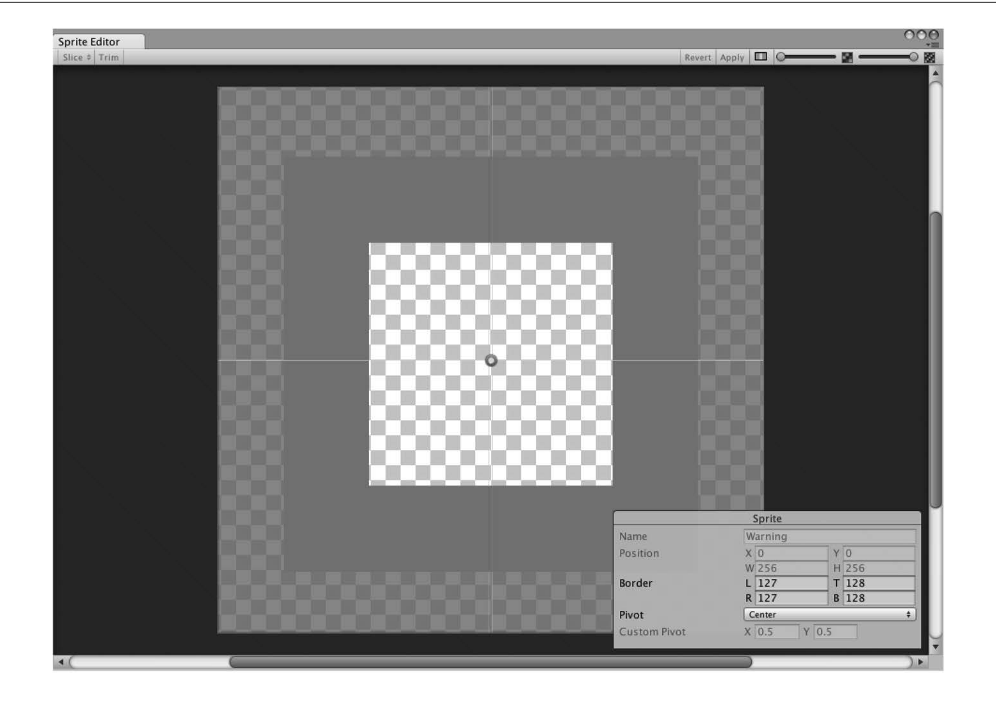


图 13-7：切割 Warning 精灵

单击 Apply 按钮。

(3) **接下来将创建 Warning UI**。这只是在 UI 上显示的一张图片，被设为拉伸至整个画面。

为了设置警告 UI，创建一个新的空游戏对象，将其命名为 Warning UI，设为 Canvas 对象的子对象。

将其锚点设为水平和垂直拉伸，将边距设为 0，使其填充整个画布。

为其添加一个 Image 组件。将该 Image 组件的 Source Image 设为刚才创建的 Warning 精灵，将 Image Type 设为 Sliced。该图片将拉伸至整个画面。

完成上述设置后，就应该进行编码了。

13.3.2 编写代码处理边界

边界对玩家是不可见的，这意味着在编辑游戏的过程中也不可见。如果想要可视化玩家能够四处飞行的空间大小，还需要再次使用 Gizmos 功能，就像处理 Asteroid Spawner 那样。

我们只关心两个同心球体，分别称为“**警告**球体”和“**销毁**球体”。这两个球体以同一个点为球心，但是具有不同的半径：警告球体的半径比销毁球体的半径小。

- 如果飞船位于警告球体内，那么没有问题，不会显示警告。
- 如果飞船位于警告球体以外，那么画面上将显示警告，告诉玩家应该掉头往回飞行。
- 如果飞船位于销毁球体之外，则游戏结束。

实际检查飞船位置的工作由 Game Manager 处理，它使用（接下来就会创建的）Boundary 对象内存储的数据来判断飞船是否在这两个球体之外。

首先创建 Boundary 对象，并添加代码来可视化这两个球体。

(1) **创建 Boundary 对象**。创建一个新的空对象，命名为 Boundary。

为该对象添加一个新的 C# 脚本，命名为 Boundary.cs，并添加下面的代码：

```
public class Boundary : MonoBehaviour {

  //当玩家距离中心这么远时，显示警告UI
  public float warningRadius = 400.0f;

  //当玩家距离中心这么远时，结束游戏
  public float destroyRadius = 450.0f;

  public void OnDrawGizmosSelected() {
    //显示一个具有警告半径的黄色球体
    Gizmos.color = Color.yellow;
    Gizmos.DrawWireSphere(transform.position,
      warningRadius);

    //显示一个具有销毁半径的红色球体
    Gizmos.color = Color.red;
    Gizmos.DrawWireSphere(transform.position,
      destroyRadius);
  }
}
```

返回到游戏编辑器时，将看到两个线框球体。黄色的球体显示了警告半径，红色球体显示了销毁半径，如图 13-8 所示。

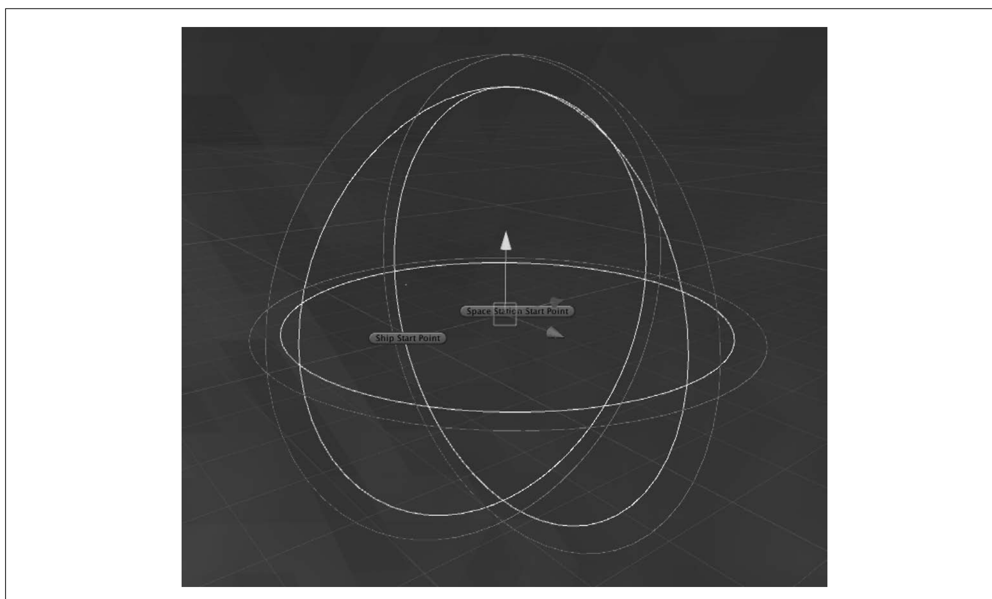


图 13-8：边界（另见彩插）



Boundary 脚本在游戏中并不实际执行自己的逻辑。相反，GameManager 使用 Boundary 对象的数据来判断玩家是否已经飞出了边界半径。

现在就创建好了边界对象，我们只需要设置 Game Manager 来使用该对象。

(2) **将边界字段添加到 GameManager 脚本，并更新 GameManager 来使用这些字段**。将下面的代码添加到 GameManager.cs：

```
public class GameManager : Singleton<GameManager> {

    //为飞船使用的预设、飞船的起始位置，以及当前的飞船对象
    public GameObject shipPrefab;
    public Transform shipStartPosition;
    public GameObject currentShip {get; private set;}

    //为空间站使用的预设、空间站的起始位置，以及当前的空间站对象
    public GameObject spaceStationPrefab;
    public Transform spaceStationStartPosition;
    public GameObject currentSpaceStation {get; private set;}

    //主摄像机上的跟随脚本
    public SmoothFollow cameraFollow;

>   //游戏的边界
>   public Boundary boundary;
```

```
    //各个UI部分的容器
    public GameObject inGameUI;
    public GameObject pausedUI;
    public GameObject gameOverUI;
    public GameObject mainMenuUI;

>   //靠近边界时显示的警告UI
>   public GameObject warningUI;

    //当前在玩游戏吗？
    public bool gameIsPlaying {get; private set;}

    //游戏的小行星生成器
    public AsteroidSpawner asteroidSpawner;

    //跟踪游戏是否被暂停
    public bool paused;

    //当游戏启动时显示主菜单
    void Start() {
      ShowMainMenu();
    }

    //显示一个UI容器，并隐藏其他所有UI容器
    void ShowUI(GameObject newUI) {

      //创建所有UI容器的列表
      GameObject[] allUI
        = {inGameUI, pausedUI, gameOverUI, mainMenuUI};

      //隐藏全部UI容器
      foreach (GameObject UIToHide in allUI) {
        UIToHide.SetActive(false);
      }

      //然后显示提供的UI容器
      newUI.SetActive(true);
    }

    public void ShowMainMenu() {
      ShowUI(mainMenuUI);

      //游戏启动时，我们还没有玩游戏
      gameIsPlaying = false;

      //也不生成小行星
      asteroidSpawner.spawnAsteroids = false;
    }

    //由被触摸的New Game按钮调用
    public void StartGame() {
      //显示游戏内UI
      ShowUI(inGameUI);

      //我们现在在玩游戏
      gameIsPlaying = true;
```

```
    //如果刚好有飞船，就销毁该飞船
    if (currentShip != null) {
      Destroy(currentShip);
    }

    //对空间站执行类似处理
    if (currentSpaceStation != null) {
      Destroy(currentSpaceStation);
    }

    //创建一个新飞船，置于起始位置
    currentShip = Instantiate(shipPrefab);
    currentShip.transform.position
      = shipStartPosition.position;
    currentShip.transform.rotation
      = shipStartPosition.rotation;

    //为空间站执行类似处理
    currentSpaceStation = Instantiate(spaceStationPrefab);

    currentSpaceStation.transform.position =
      spaceStationStartPosition.position;

    currentSpaceStation.transform.rotation =
      spaceStationStartPosition.rotation;

    //使跟随脚本跟踪新飞船
    cameraFollow.target = currentShip.transform;

    //开始生成小行星
    asteroidSpawner.spawnAsteroids = true;

    //使生成器对准新的空间站
    asteroidSpawner.target = currentSpaceStation.transform;

  }

  //由被摧毁时结束游戏的对象调用
  public void GameOver() {
    //显示Game Over UI
    ShowUI(gameOverUI);

    //我们不再玩游戏
    gameIsPlaying = false;

    //销毁飞船和空间站
    if (currentShip != null)
      Destroy (currentShip);

    if (currentSpaceStation != null)
      Destroy (currentSpaceStation);

>   //如果警告UI是可见的，就停止显示警告UI
>   warningUI.SetActive(false);
```

```
      //停止生成小行星
      asteroidSpawner.spawnAsteroids = false;

      //移除游戏中所有残存的小行星
      asteroidSpawner.DestroyAllAsteroids();
    }

    //当触摸Pause或Resume按钮时调用
    public void SetPaused(bool paused) {

      //在In-Game UI和暂停UI之间切换
      inGameUI.SetActive(!paused);
      pausedUI.SetActive(paused);

      //如果已暂停……
      if (paused) {
        //停止时间
        Time.timeScale = 0.0f;
      } else {
        //恢复时间
        Time.timeScale = 1.0f;
      }
    }

>   public void Update() {
>
>     //如果没有飞船，就返回
>     if (currentShip == null)
>       return;
>
>     //如果飞船位于边界的销毁半径之外，则游戏结束
>     //如果在销毁半径之内，但是位于警告半径之外，则显示警告UI
>     //如果在警告半径之内，则不显示警告UI
>
>     float distance =
>       (currentShip.transform.position
>         - boundary.transform.position)
>           .magnitude;
>
>     if (distance > boundary.destroyRadius) {
>       //飞船超出了销毁半径，因此游戏将结束
>       GameOver();
>     } else if (distance > boundary.warningRadius) {
>       //飞船超出了警告半径，因此显示警告UI
>       warningUI.SetActive(true);
>     } else {
>       //飞船位于警告阈值内，因此不显示警告UI
>       warningUI.SetActive(false);
>     }
>
>   }

  }
```

新代码使用刚才创建的 Boundary 类，检查玩家是否飞出了警告半径或销毁半径。每一帧都会检查玩家与边界球体中心的距离。如果超出警告半径，则显示警告 UI。如果超出销毁半径，则结束游戏。如果玩家在警告半径**之内**，则没有问题，从而禁用警告半径。这意味着如果玩家飞出警告半径，然后又返回安全区域，则会看到警告 UI 显示后又消失。

接下来只需要填充各输入框。Game Manager 需要引用刚才创建的 Boundary 对象，还要引用 Warning UI。

(3) **配置 Game Manager 来使用边界对象**。将 Warning UI 拖放到 Warning UI 框，将 Boundary 对象拖放到 Boundary 框。
(4) **测试游戏**。靠近边界时，警告将会显示。如果不掉头回到安全区域，游戏将会结束！

13.4 最终优化

祝贺你！现在你已经设置好了一个相当复杂的太空射击游戏的核心玩法。通过前几节的动手操作，你设置了一个太空环境，创建了飞船、空间站、小行星和激光束，设置好了对象的物理计算，并设置了各种逻辑组件来把各个对象连接起来。在此基础上，你创建了 UI，使得在 Unity 编辑器之外玩游戏成为可能。

游戏的核心已经完成，但是还有视觉优化的空间。因为这个游戏的画面比较空旷，所以我们没有太多视觉参照点来让玩家有高速移动感。另外，我们还将为飞船和小行星添加轨迹渲染器，使游戏更加丰富。

13.4.1 太空尘埃

如果你以前玩过空战游戏，例如《自由枪骑兵》或《独立战争》，那么可能会注意到，当玩家四处飞行时，小尘埃、残骸和其他小物体会从玩家旁边飞过。

为了改进我们的游戏，我们将添加小尘埃，当玩家飞过时，视觉上有一种深度感和透视感。我们将使用一个粒子系统实现此效果，粒子系统会跟随玩家移动，在一个围绕玩家的球体上不断创建尘埃粒子。重要的是，这些尘埃粒子不会相对玩家移动。这意味着当玩家飞行时，尘埃粒子将会出现，然后玩家在飞行时将经过它们。这就在游戏中更好地创建了一种速度感。

创建尘埃粒子需要执行下面的步骤。

(1) **将 Ship 预设拖放到场景中**。我们将对预设做一些修改。
(2) **创建 Dust 子对象**。创建一个新的空游戏对象，命名为 Dust，设为刚才拖出的 Ship 游戏对象的子对象。
(3) **为其添加一个 Particle System 组件**。复制图 13-9 中显示的设置。

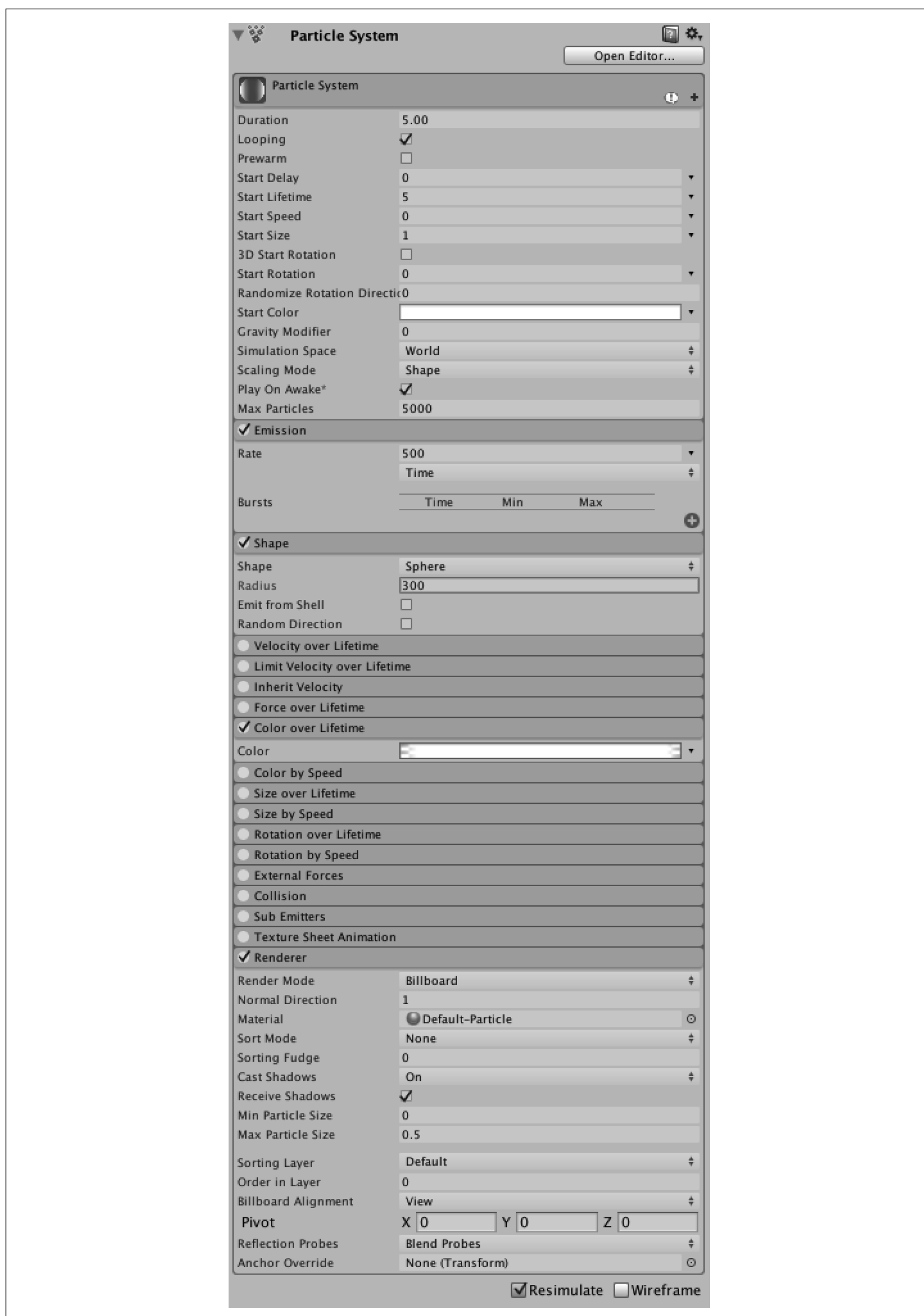


图 13-9：尘埃粒子的设置

这个粒子系统的关键部分在于，Simulation Space 设置为 `World`，Shape 为 `Sphere`。通过将 Simulation Space 设为 `World`，粒子不会随着其父对象（Ship）移动，这意味着飞船将飞过粒子。

(4) **对预设应用修改**。选择 Ship 对象，然后单击 Inspector 顶部的 Apply 按钮。这将把做出的修改保存到预设。我们还没有完成对飞船的设置，所以还不要删除它。

在图 13-10 中可以看到粒子系统的应用。在天空盒相对平滑的颜色上，粒子系统创建了一种星空感。

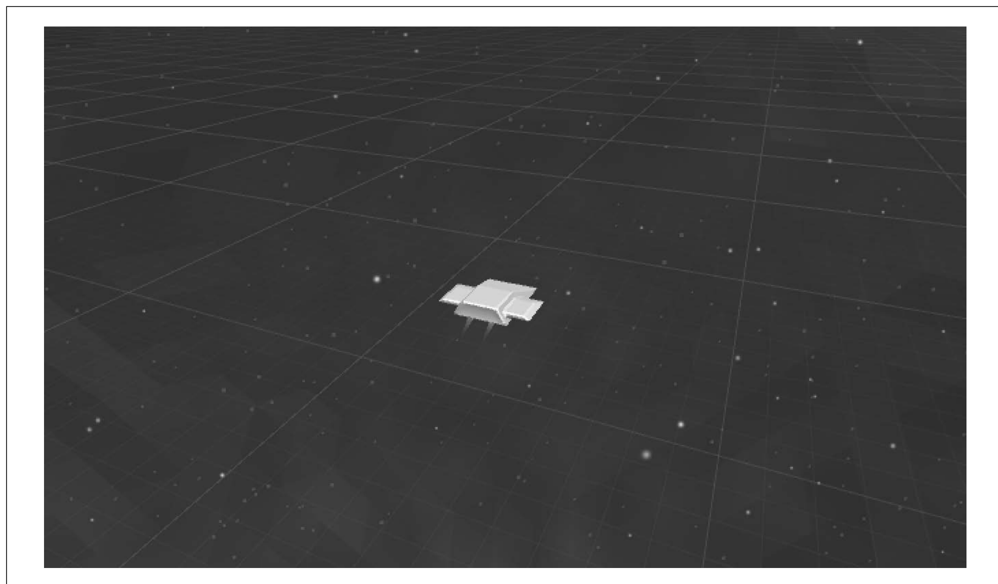


图 13-10：尘埃粒子系统

13.4.2 轨迹渲染器

飞船的模型非常简洁，但是没有理由不能用一些特效来进行美化。我们将为飞船添加两个线渲染器，创建出飞船尾部引擎的效果。

(1) **为轨迹创建一个新的 Material**。打开 Assets 菜单，选择 Create → Material。将新材质命名为 Trail，放到 Objects 文件夹中。

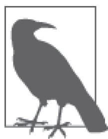
(2) **使 Trail 材质使用 Additive 着色器**。选择 Trail 材质，将其 Shader 改为 Mobile → Particles → Additive。这是一个简单的着色器，只是将其颜色加到背景上。保留 Particle Texture 为空，我们不需要使用这个设置。

(3) **为 Ship 添加一个新的子对象**。将其命名为 Trail 1，放到 (−0.38, 0, −0.77) 位置。

(4) **添加一个 Trail Renderer 组件**。使用图 13-11 所示的设置。注意其使用的 Material 为刚刚创建的 Trail 材质。

Trail Renderer	
Cast Shadows	On
Receive Shadows	✓
Motion Vectors	Per Object Motion
Materials	
Size	1
Element 0	Trail
Lightmap Parameters	None (Lightmap Parameters)
Time	0.25
Min Vertex Distance	0.1
Autodestruct	☐
Width	0.5
Color	
Corner Vertices	0
End Cap Vertices	0
Alignment	View
Texture Mode	Stretch
Generate Lighting Data	☐
Sorting Layer	Default
Order in Layer	0
Light Probes	Blend Probes
Reflection Probes	Blend Probes
Anchor Override	None (Transform)

图 13-11：飞船的 Trail Renderer 设置



Trail Render 渐变使用的颜色为：

- #000B78FF
- #061EF9FF
- #0080FCFF
- #000000FF
- #00000000

你会注意到，越接近飞船，颜色就越深。由于 Trail 材质使用了 Additive 着色器，可以让轨迹有淡出效果。

(5) **复制对象**。设置好第一个轨迹以后，打开 Edit 菜单并选择 Duplicate，复制该轨迹。将复制得到的对象移动到 (0.38, 0, −0.77)。



第二个轨迹的位置与第一个轨迹相同，只不过翻转了 x 分量。

(6) **将做出的修改应用到预设**。选择 Ship 对象，单击 Inspector 顶部的 Apply 按钮。最后，从场景中删除 Ship。

现在就可以进行测试了。驾驶飞船时，飞船后面将出现两条蓝色轨迹，如图 13-12 所示。

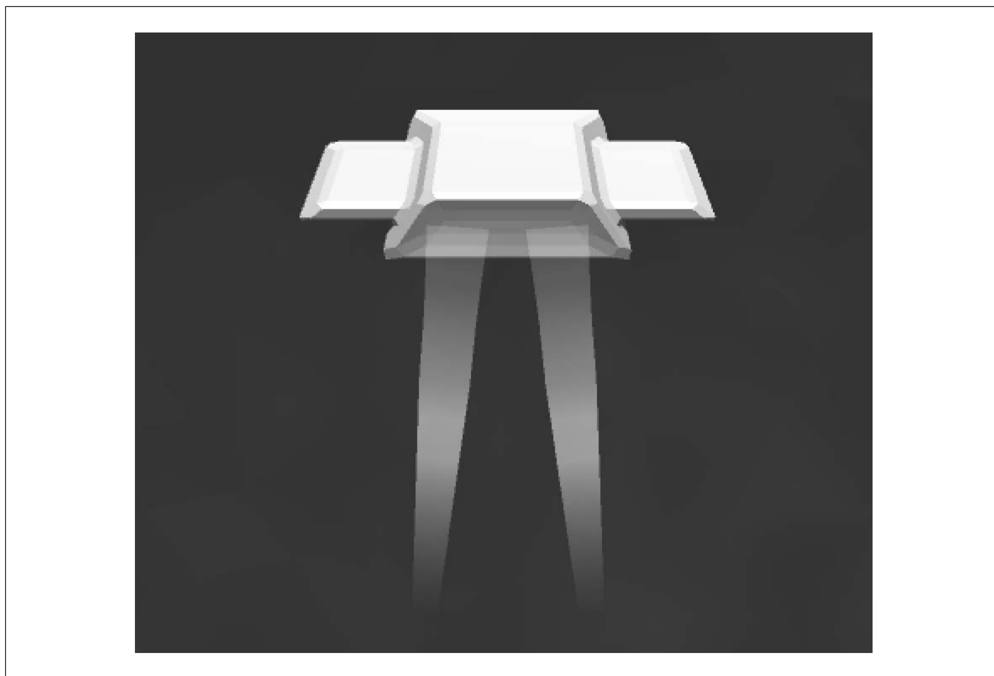


图 13-12：飞船后面的引擎轨迹

现在，我们为小行星应用类似的效果。游戏中的小行星颜色很暗，虽然有指示器来帮助玩家了解它们的位置，但是为它们加上颜色会更好。为了改进效果，我们将为小行星添加一个轨迹渲染器。

(1) **在场景中添加一个 Asteroid**。将 Asteroid 预设拖放到场景中，以便进行修改。

(2) **为其 Graphics 子对象添加一个 Trail Renderer 组件**。使用如图 13-13 所示的设置。

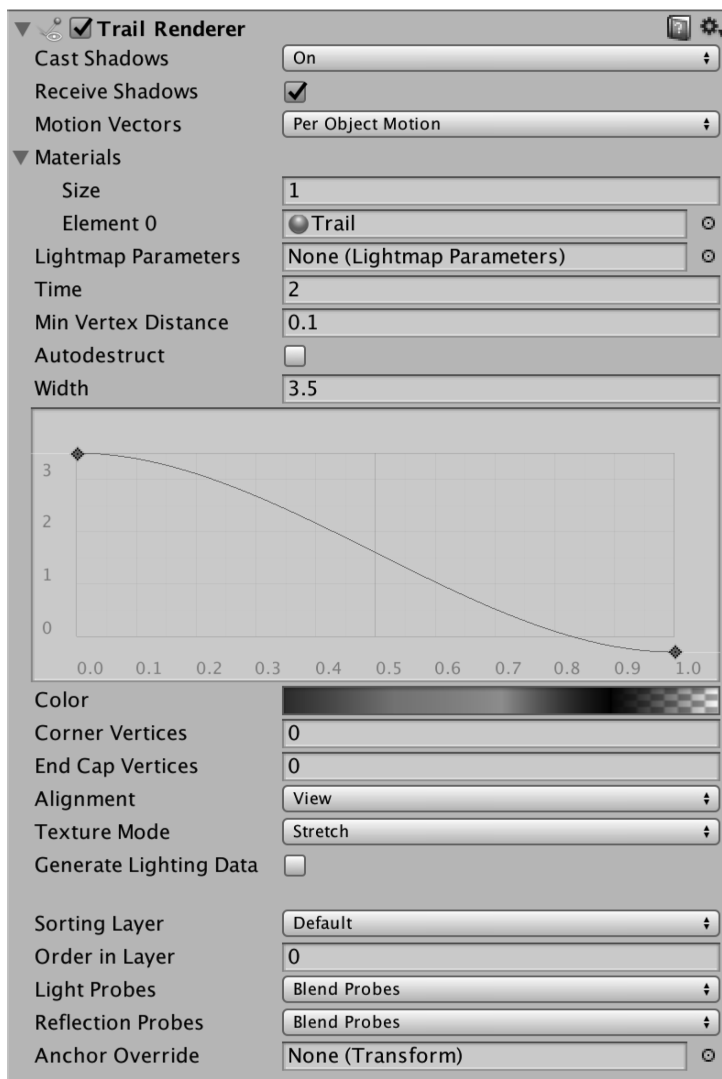


图 13-13：小行星的轨迹设置

(3) 将修改应用到 Asteroid 预设，然后从场景中移除该预设。

现在，小行星后面将带有一个明亮的轨迹。图 13-14 显示了正在进行中的游戏。



图 13-14：正在进行中的游戏（另见彩插）

13.4.3 音效

我们还需要添加最后一个部分：音效！虽然在真实的太空中是没有声音的，但是通过添加音效，电子游戏的效果将大大增强。要添加的音效有 3 种：飞船发动机的轰鸣、激光束的滋滋声，以及小行星的爆炸声。我们将一次添加一种。



本书的文件中包含了一些公共领域的音效文件，放在 Audio 文件夹中。

1. 飞船

首先为飞船添加一种循环播放的音效。

(1) **将 Ship 添加到场景中**。我们将对其做一些修改。
(2) **为 Ship 添加一个 Audio Source 组件**。通过 Audio Source 才能够添加音效。
(3) **打开 Loop 设置**。在飞船的飞行过程中，我们想让引擎声一直播放。
(4) **添加飞船发动机声效**。将 Engine 音频片段拖放到 AudioClip 框中。
(5) **保存对预设的修改**。引擎声就添加完成了。

添加循环音效极其简单，只需要做很少的工作，游戏在整体上就能够得到巨大提升。



先不要从场景中删除 Ship，稍后还将为其添加更多内容。

2. 武器音效

添加武器音效要复杂一些。我们想在每次发射武器时播放音效，这意味着需要让代码知道音效的存在。

首先需要为两个武器发射点添加音频源。

(1) **为武器发射点添加 Audio Source**。同时选中两个武器发射点，然后添加 Audio Source。
(2) **将 Laser 效果添加到音频源**。完成之后，关闭 Play On Awake 设置，这是因为我们希望仅当发射武器时才播放声音。
(3) **添加代码，当发射武器时播放音效**。向 ShipWeapons.cs 添加下面的代码：

```
public class ShipWeapons : MonoBehaviour {

  //为每个武器使用的预设
  public GameObject shotPrefab;

  public void Awake() {
    //当此对象启动时，告诉InputManager使用自己作为当前的武器对象
    InputManager.instance.SetWeapons(this);
  }

  //移除对象时调用
  public void OnDestroy() {
    //如果没在玩游戏，就不做此处理
    if (Application.isPlaying == true) {
      InputManager.instance.RemoveWeapons(this);
    }
  }

  //武器发射位置的列表
  public Transform[] firePoints;

  //发射武器的firePoints的索引
  private int firePointIndex;

  //由InputManager调用
  public void Fire() {

    //如果没有武器发射点，就返回
    if (firePoints.Length == 0)
      return;

    //计算出从哪个发射点发射
    var firePointToUse = firePoints[firePointIndex];

    //在发射点位置使用其旋转创建新武器
    Instantiate(shotPrefab,
      firePointToUse.position,
      firePointToUse.rotation);

>   //如果发射点有音频源组件，就播放音效
>   var audio
>     = firePointToUse.GetComponent<AudioSource>();
```

```
>      if (audio) {
>        audio.Play();
>      }

       //移动到下一个发射点
       firePointIndex++;

       //如果在到达列表中的最后一个发射点以后继续移动，就回到队列的开始位置
       if (firePointIndex >= firePoints.Length)
         firePointIndex = 0;
     }

   }
```

这段代码检查武器的发射点是否有 `AudioSource` 组件。如果有，就播放发射武器的声音。

(4) **保存对 Ship 预设的修改，然后从场景中移除它。**

现在就完成了对武器音效的设置。每次发射武器时，都将听到音效。

13.4.4 爆炸

还需要添加最后一种音效：当发生爆炸时，播放爆炸音效。添加这个音效很简单：只需要为爆炸对象添加一个音频源，并将其设为 Play On Awake。当发生爆炸时，就会自动播放 Explosion 声音。

(1) **在场景中添加 Explosion。**
(2) **为爆炸添加一个 Audio Source。**拖入 Explosion 音频片段，并打开 Play On Awake。
(3) **保存对预设的修改，并从场景中移除该预设。**

现在，每当发生爆炸时，就会听到爆炸音效！

13.5 小结

祝贺你！ *Rockfall* 游戏现在已经完成。你可以决定接下来如何处理。

关于接下来的操作，有如下建议。

添加新武器
: 可以添加一个火箭炮，让它转动以瞄准目标。

添加敌人来攻击玩家
: 小行星很简单，它们径直飞向空间站，并不会追击玩家。

为空间站添加毁坏效果
: 添加一个粒子系统，当小行星击中空间站时，在碰撞点发射烟尘和火焰。这种效果并不符合实际，但是我们在游戏中添加的许多其他功能也不是在现实世界中实际存在的。

第四部分

高级功能

本部分将更深入地讨论 Unity 的一些具体功能，不仅更加详细地探索 UI 系统，还将拓展编辑器，为其增添额外的功能。本部分还将介绍光照和着色系统，以及庞大的 Unity 生态系统。最后将讨论如何将你的游戏发布到设备上，供全世界的玩家体验。

第 14 章

光照与着色器

本章将介绍光照与材质，它们是除纹理之外决定游戏外观效果的首要因素。具体来说，我们将深入介绍 Standard 着色器，它简化了创建美观材质的过程。另外还将介绍如何编写自己的着色器，从而在很大程度上自己控制对象在游戏中的显示效果。最后讨论如何使用全局光照和光照贴图，通过在场景中对光线传播进行真实建模，创建看起来很棒的环境。

14.1 材质与着色器

在 Unity 中，对象的外观由其附加的**材质**所定义。材质由两个元素构成：**着色器**，以及着色器使用的数据。

着色器是在显卡上运行的一个非常小的程序。你在屏幕上看到的每个东西，都是着色器为每个像素计算出正确的颜色值后的显示结果。

Unity 中主要有两种不同类型的着色器，即**表面着色器**和**顶点－片段着色器**。

表面着色器负责计算对象表面的颜色。表面的颜色由多个组件定义，包括其反射率、平滑性等。表面着色器的工作是为对象的每个像素计算这些属性的值，然后表面信息被返回到 Unity，Unity 把表面信息与场景中每个灯光的信息结合起来，决定像素的最终光照颜色。

顶点－片段着色器则简单得多。这种着色器负责计算像素的最终颜色，如果着色器需要包含光照信息，那么需要你自己计算。顶点－片段着色器提供了低级控制，意味着它们能够实现出色的效果。因为这种着色器一般更加简单，所以相比于表面着色器，速度通常也更快。



实际上，Unity 会把表面着色器编译为顶点－片段着色器。它替你完成了困难的工作，实现必要的光照计算来得到真实的光照效果。表面着色器能实现的任何效果，在顶点－片段着色器中也能实现，只不过需要更多的工作。

除非有特殊的用例需要，否则表面着色器一般是最好的选择。这两种着色器本章都会探讨。



Unity 还提供了第三种类型的着色器，称为**固定功能着色器**。固定功能着色器将预定义操作组合到一起，而不是让你编写自定义着色器。在自定义着色器得到广泛应用之前，固定功能着色器是主要采用的着色器；它们比自定义着色器更加简单，但是效果则不如自定义着色器，现在也不建议使用它们。本章不会讨论固定功能着色器，不过如果你确实想学习它们，可以查阅 Unity 的文档，其中有一篇教程介绍了如何编写固定功能着色器（https://docs.unity3d.com/Manual/ShaderTut1.html）。

我们首先创建一个自定义表面着色器，它与标准着色器很相似，但是添加了显示边缘高光（即高光显示对象边缘）的能力。图 14-1 显示了这种效果的一个例子。

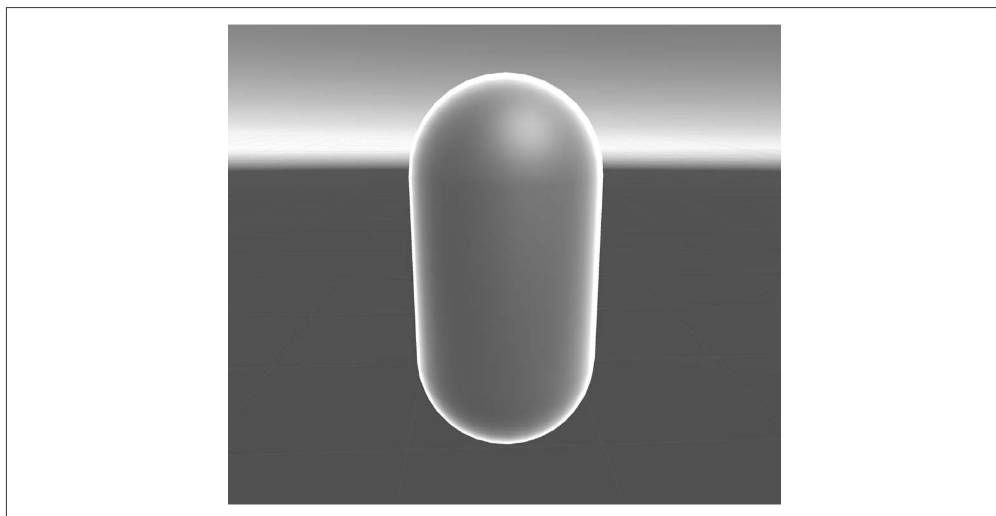


图 14-1：使用自定义着色器的边缘高光

创建该效果需要执行下面的步骤。

(1) **创建一个新项目**。按照自己的想法任意命名此项目，然后选择 3D 模式。

(2) 选择 Create → Shader → Surface Shader，**创建一个新着色器**。将新着色器命名为 SimpleSurfaceShader。

(3) **双击该着色器**。

(4) **替换为下面的代码**：

```
Shader "Custom/SimpleSurfaceShader" {

    Properties {
        //使用此颜色为对象着色
        _Color ("Color", Color) = (0.5,0.5,0.5,1)

        //对象的纹理
```

```
    //默认为纯白色纹理
    _MainTex ("Albedo (RGB)", 2D) = "white" {}

    //表面的平滑度
    _Smoothness ("Smoothness", Range(0,1)) = 0.5

    //表面的金属感程度
    _Metallic ("Metallic", Range(0,1)) = 0.0

}

SubShader {
    Tags { "RenderType"="Opaque" }
    LOD 200

    CGPROGRAM
        //基于物理的标准光照模型，在所有灯光类型上启用了阴影
        #pragma surface surf Standard fullforwardshadows

        //使用着色器模型3.0 target来获得更美观的光照效果
        #pragma target 3.0

        //下面的变量是“统一的”——为每个像素使用相同的值

        //用于反射率的纹理
        sampler2D _MainTex;

        //用于着色反照率的颜色
        fixed4 _Color;

        //光滑性和金属感属性
        half _Smoothness;
        half _Metallic;

        //Input包含的变量值对于每个像素均不同
        struct Input {
            //此像素的纹理坐标
            float2 uv_MainTex;

        };

        //此函数计算此表面的属性
        void surf (Input IN,
          inout SurfaceOutputStandard o) {

            //使用IN中存储的数据和上面的变量计算值，然后存储到o中

            //反照率来自用color着色的纹理
            fixed4 c =
              tex2D (_MainTex, IN.uv_MainTex) * _Color;
            o.Albedo = c.rgb;

            //金属感和平滑性来自滑动条变量
            o.Metallic = _Metallic;
            o.Smoothness = _Smoothness;
```

```
                //Alpha值来自我们为反照率使用的纹理
                o.Alpha = c.a;

            }
        ENDCG
    }

    //如果运行此着色器的计算机没有运行着色器模型3.0的能力，
    //则使用内置的Diffuse着色器。这种着色器的效果较差，
    //但是肯定是能够工作的
    FallBack "Diffuse"
}
```

(5) **创建一个新材质**，命名为 SimpleSurface。

(6) **选择新材质**，打开 Inspector 顶部的 Shader 菜单。选择 Customer → SimpleSurfaceShader。

现在，表面着色器的属性将显示在 Inspector 中，如图 14-2 所示。

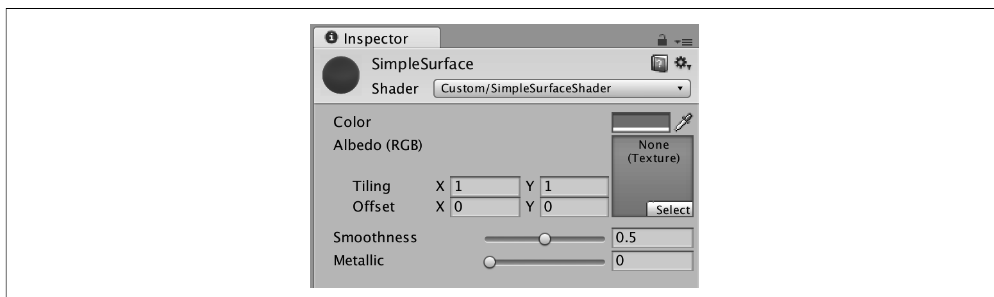


图 14-2：自定义着色器的 Inspector

(7) 选择 GameObject → 3D Object → Capsule，**创建一个新的胶囊体**。

(8) **将 SimpleShader 材质拖放到胶囊体上**，它将开始使用新材质，如图 14-3 所示。

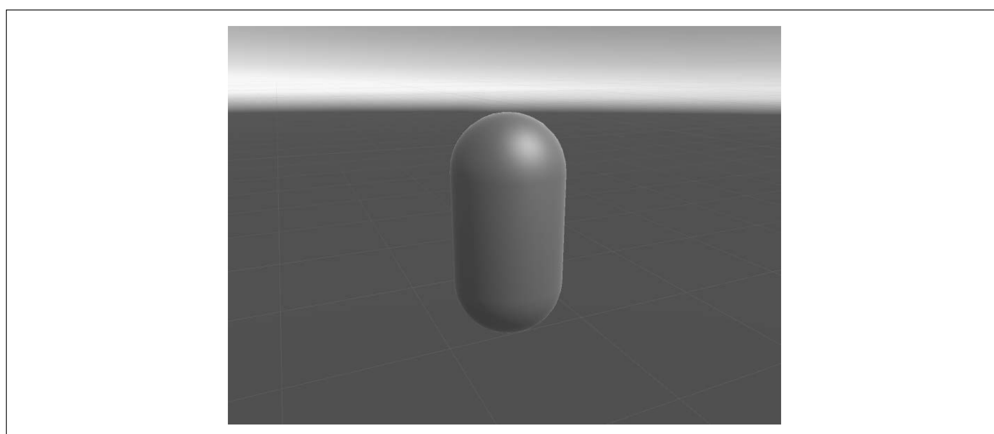


图 14-3：使用自定义着色器的胶囊体

现在，该对象看起来与标准着色器十分相似。接下来就添加边缘高光。

为了计算边缘高光，需要知道以下 3 个值：

- 光照的颜色
- 边缘的厚度
- 摄像机指向的方向与表面指向的方向之间的夹角



表面指向的方向称为该表面的**法线**（normal）。我们将要编写的代码会使用这个术语。

前两项是**统一的**，即它们的值应用到对象的每个像素上。第三项是**变化的**，意味着其值取决于你在看什么地方——摄像机方向与表面法线之间的夹角取决于你是在查看圆柱体的中间还是边缘。

显卡将在运行时计算变化值，供表面着色器使用，而统一值则作为材质的属性提供，可通过 Inspector 进行修改。因此，为了添加对边缘高光的支持，我们首先需要为着色器添加两个统一变量。

(1) **修改着色器的 `Properties` 节**，以包含下面的代码：

```
Properties {
    //使用此颜色为对象着色
    _Color ("Color", Color) = (0.5,0.5,0.5,1)

    //对象的纹理
    //默认为纯白色纹理
    _MainTex ("Albedo (RGB)", 2D) = "white" {}

    //表面的平滑度
    _Smoothness ("Smoothness", Range(0,1)) = 0.5

    //表面的金属感程度
    _Metallic ("Metallic", Range(0,1)) = 0.0

>   //边缘高光的颜色
>   _RimColor ("Rim Color", Color) = (1.0,1.0,1.0,0.0)
>
>   //边缘高光的厚度
>   _RimPower ("Rim Power", Range(0.5,8.0)) = 2.0

}
```

这段代码使着色器在 Inspector 中显示两个新字段。现在，我们需要使这两个属性对着色器代码可用，以便 `surf` 函数能够使用它们。

(2) **为着色器添加下面的代码**：

```
//平滑性和金属感属性
half _Smoothness;
```

```
    half _Metallic;

> //边缘高光的颜色
> float4 _RimColor;
>
> //边缘高光的厚度，值越接近0，表示边缘越厚
> float _RimPower;
```

接下来，我们需要让着色器能够获知摄像机的方向。着色器使用的全部变化值都包含在 `Input` 结构中，这意味着需要在该结构中添加观看方向。

在 `Input` 结构中可以添加多个字段，Unity 将自动用相关信息填充它们。如果添加一个 `float3` 类型的变量，并命名为 `viewDir`，那么 Unity 将把摄像机的指向存储到该变量中。



`viewDir` 不是 Unity 为变化信息自动使用的唯一变量名。如果想了解这些变量名的完整列表，请查阅 Unity 的 Surface Shader 文档（https://docs.unity3d.com/Manual/SL-SurfaceShaders.html）。

(3) **向 `Input` 结构添加下面的代码：**

```
    struct Input {
        //此像素的纹理坐标
        float2 uv_MainTex;

>       //摄像机观看此顶点的方向
>       float3 viewDir;
    };
```

材质的 Inspector 现在将显示新添加的字段，如图 14-4 所示。

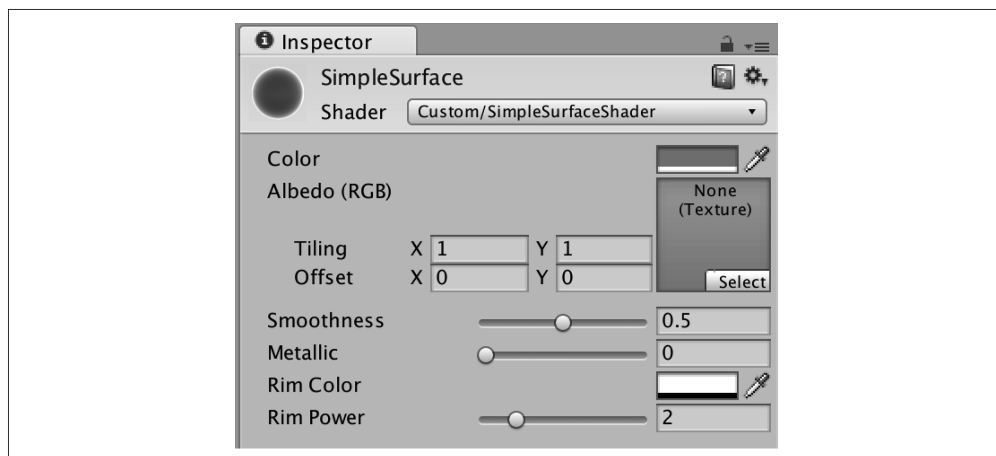


图 14-4：显示新添加字段的 Inspector

现在，我们就有了计算边缘高光所需的全部信息；最后一步是实际执行计算，并将其添加到表面的信息中。

(4) **将下面的代码添加到 `surf` 函数：**

```
// 此函数计算此表面的属性
void surf (Input IN, inout SurfaceOutputStandard o) {

    //使用IN中存储的数据和上面的变量计算值，然后存储到o中

    //反照率来自用color着色的纹理
    fixed4 c = tex2D (_MainTex, IN.uv_MainTex) * _Color;
    o.Albedo = c.rgb;

    //金属性和平滑性来自滑动条变量
    o.Metallic = _Metallic;
    o.Smoothness = _Smoothness;

    //alpha值来自我们为反照率使用的纹理
    o.Alpha = c.a;

>   //计算边缘光在此像素的亮度
>   half rim =
>     1.0 - saturate(dot (normalize(IN.viewDir), o.Normal));
>
>   //使用此亮度计算边缘颜色，将其用于发射光
>   o.Emission = _RimColor.rgb * pow (rim, _RimPower);

}
```

(5) **保存着色器，并返回 Unity**。胶囊体现在就有了边缘高光！结果如图 14-5 所示。

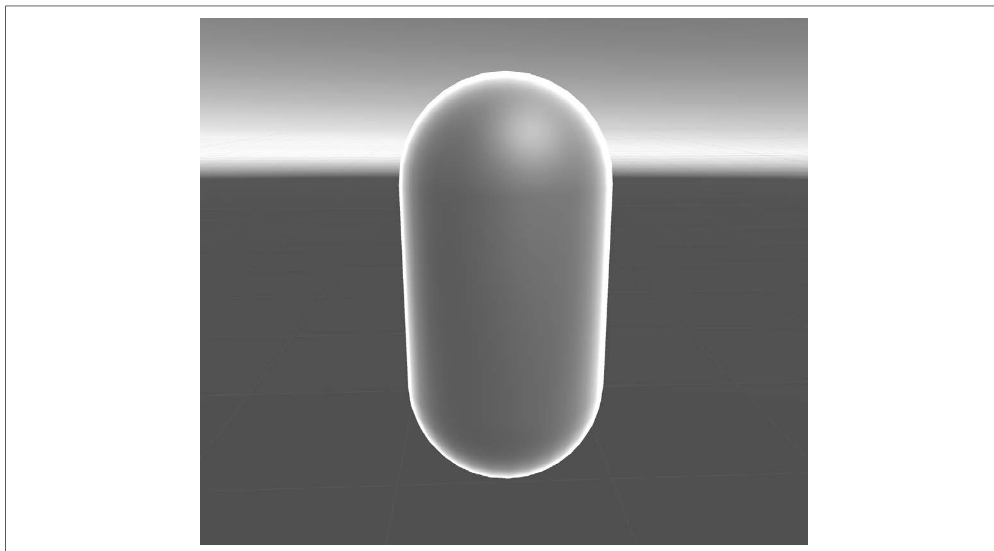


图 14-5：带有边缘高光的胶囊体

还可以通过材质的属性调整边缘高光。试着修改 Rim Color 设置来改变边缘高光的亮度和颜色，修改 Rim Power 设置来调整边缘的厚度。

表面着色器很适合拓展已有的着色系统，而且如果想让表面响应自己添加到场景中的灯光，它们是最佳选择。但是在有些情况下，你并不关心光照，或者需要精细控制表面的外观。此时，可以创建完全自定义的片段－顶点着色器。

片段–顶点（无光照）着色器

片段－顶点着色器的命名缘于它是两个着色器合二为一：**片段着色器**和**顶点着色器**。这是两个独立的功能，控制着表面的渲染方式。

顶点着色器将每个**顶点**（即对象表面上的每个点）从世界空间转换为视空间，以便为渲染做好准备。世界空间意味着你在 Unity 编辑器中看到的世界：对象被放置在世界空间中，你可以四处移动它们。但是，当 Unity 需要使用摄像机渲染场景的时候，摄像机必须先把场景中每个对象的位置转换到**视空间**：在这个空间中，整个世界以及其中的每个对象的位置将被重新设定，使摄像机处在世界的中心。另外，在视空间中，整个世界将被重新塑造，使远离摄像机的对象变得更小。顶点着色器还负责计算应该传递给片段着色器的变化变量的值。



你几乎用不着编写自己的顶点着色器，但是在一些情况中，自己编写会很有帮助。例如，如果想扭曲一个对象的形状，就可以编写自己的顶点着色器来修改每个顶点的位置。

片段着色器是这对着色器组合的另一半，负责计算对象的每个片段（即像素）的最终颜色。片段着色器接受顶点着色器计算出的变化变量的值，该值将根据被渲染的片段与其最接近的顶点之间的临近性进行**插值**或混合。

因为片段着色器完全控制对象的最终颜色，所以需要由着色器自身来计算附近灯光的效果。如果着色器自己不执行计算，那么表面就不会呈现光照效果。

由于这个原因，建议使用表面着色器来使表面呈现光照效果。光照计算可能非常复杂，如果不必考虑这个问题的话，工作会简单许多。



表面着色器实际上就是片段－顶点着色器。Unity 会替你将表面着色器转换为低级的片段－顶点代码，并添加光照计算。

这样做的缺点在于，表面着色器是为一般用例设计的，可能比手工编写的着色器效率低一些。

为了演示片段－顶点着色器的工作方式，我们将创建一个简单的片段－顶点着色器，将对象渲染为单一的亚光色。然后，我们将修改着色器，根据对象在屏幕上的位置来渲染渐变。

(1) 打开 Assets 菜单，选择 Create → Shader → Unlit Shader，**创建一个新着色器**。将新着色器命名为 SimpleUnlitShader。

(2) **双击打开该着色器**。

(3) 用下面的代码替换文件的内容：

```
Shader "Custom/SimpleUnlitShader"
{
    Properties
    {
        _Color ("Color", Color) = (1.0,1.0,1.0,1)

    }
    SubShader
    {
        Tags { "RenderType"="Opaque" }
        LOD 100

        Pass
        {
            CGPROGRAM

            //定义在此着色器中应该使用什么函数

            //vert函数将用作顶点着色器
            #pragma vertex vert

            //frag函数将用作片段着色器
            #pragma fragment frag

            //包含Unity中的大量实用工具
            #include "UnityCG.cginc"

            float4 _Color;

            //将此结构提供给每个顶点的顶点着色器
            struct appdata
            {
                //顶点在世界空间中的位置
                float4 vertex : POSITION;

            };

            //将此结构提供给每个片段的片段着色器
            struct v2f
            {
                //片段在屏幕空间中的位置
                float4 vertex : SV_POSITION;
            };

            //给定顶点时，进行转换
            v2f vert (appdata v)
            {
                v2f o;

                //将顶点乘以Unity提供的矩阵（来自UnityCG.cginc），将其从世界空间
                //转换到视图空间
                o.vertex = UnityObjectToClipPos(v.vertex);
```

```
            //返回并传递给片段着色器
            return o;
        }

        //给定关于临近顶点的插值信息，返回最终的颜色
        fixed4 frag (v2f i) : SV_Target
        {
           fixed4 col;

            //渲染提供的颜色
            col = _Color;

            return col;
        }
        ENDCG
    }
  }
}
```

(4) 打开 Assets 菜单，选择 Create → Material，**创建一个新材质**。将新材质命名为 SimpleShader。

(5) **选择新材质**，将着色器改为 Custom → SimpleUnlitShader。

(6) 打开 GameObject 菜单，选择 3D Object → Sphere，**在场景中创建一个球体**。将 SimpleShader 材质拖放到该球体上。

现在球体将呈现单一的亚光色，结果如图 14-6 所示。

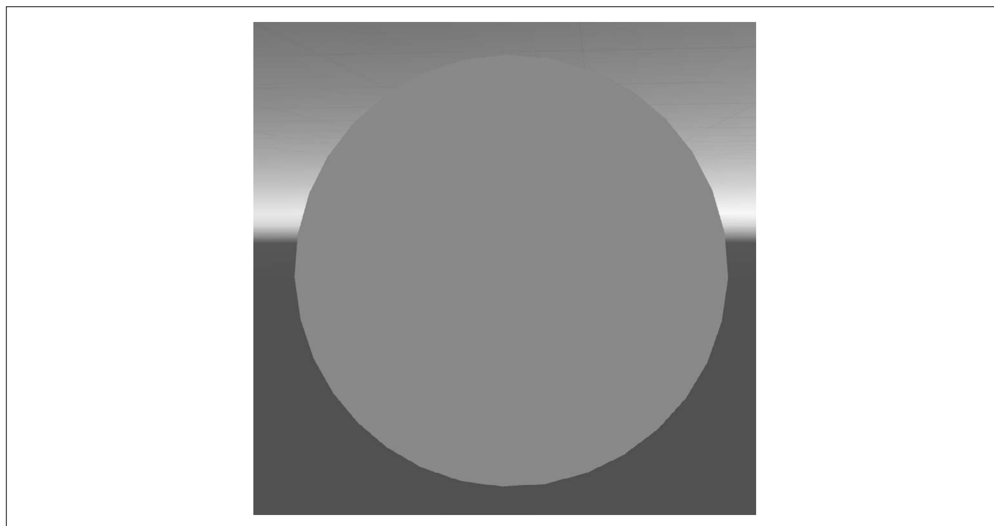


图 14-6：使用亚光色渲染后的球体（另见彩插）



用亚光色着色一个对象是很常见的任务，因此 Unity 自身捆绑了一个着色器，与我们刚才编写的非常相似。在 Shader 菜单的 Unlit → Color 子菜单下可找到该着色器。

接下来将扩展这个着色器，使其发生动态变化。这并不需要编写脚本，所有动画将在图形着色器内完成。

(1) **向 `frag` 函数添加下面的代码**：

```
fixed4 frag (v2f i) : SV_Target
{
   fixed4 col;

   //渲染提供的颜色
   col = _Color;

>  //随时间淡出——起初是黑色，逐渐变为_Color
>  col *= abs(_SinTime[3]);

   return col;
}
```

(2) **返回 Unity**，注意对象已经变为黑色。这是预期的效果。

(3) **单击 Play 按钮**，观察对象淡入并淡出。图 14-7 显示了这种效果的一个例子。

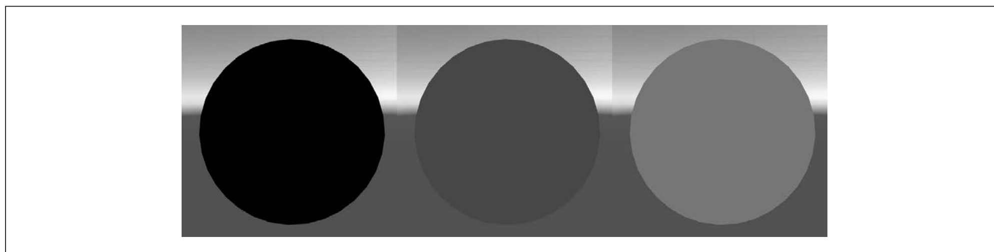


图 14-7：对象淡入并淡出（另见彩插）

片段 – 顶点着色器让我们对对象的外观具有很大的控制权，在本节中我们已经感受到了这一点。全面讨论着色器会花费整本书的篇幅，如果你想深入了解如何使用着色器，可查阅 Unity 的详尽文档（https://docs.unity3d.com/Manual/SL-Reference.html）。

14.2　全局光照

当一个对象被照亮时，负责渲染该对象的着色器需要执行一些复杂的计算，以确定该对象收到的光照量，然后使用这些信息计算摄像机能够看到的对象颜色。这么做通常没有问题，但是有些信息很难在运行时计算。

例如，如果一个球体位于一个白色的表面上，并受到阳光直射，那么应该下面也有光照亮球体，因为光是从地面向上反射的。但是，着色器只能知道太阳自身的方向，因此不会显示这种反射光的效果。这种效果肯定是可以计算出来的，但是要在每一帧中解决这个问题，很快就成为了一种挑战。

更好的解决方法是使用**全局光照**和**光照贴图**。全局光照指的是许多彼此相关的技术，它们计算出场景中每个表面接受的光照量，并考虑光线如何在对象上反射。

全局光照能够得到很真实的光照效果，但是非常占用处理器。因此，也可以在 Unity 编辑器中提前完成光照计算，并将结果保存到**光照贴图**中。光照贴图记录了场景中每个表面的每个部分最终收到的光照量。

由于全局光照计算是提前完成的，所以只能考虑肯定不会移动的对象（因而会改变灯光在场景中的工作方式）。游戏中的任何移动对象都不能直接使用全局光照，而是需要使用稍后将讨论的一种不同的解决方案。

使用光照贴图能够显著提升场景中真实光照的性能，因为光照计算已经提前完成并被保存到了纹理中。但是，如果使用光照贴图，就必须把这些纹理加载到内存中，以便渲染器使用。如果场景已经很复杂，或者已经使用了大量纹理，这可能会导致问题。为了减轻负担，一种方法是降低光照贴图的分辨率，但是这也会降低光照的视觉质量。

了解这一点之后，我们通过创建一个场景来深入讨论如何使用全局光照。我们首先设置一些具有不同颜色的材质，以帮助查看光线在场景中如何反射。然后，创建一些对象，让其使用全局光照系统。

(1) **在 Unity 中创建一个新场景**。

(2) **创建一个新材质**，命名为 Green。保留 Standard 着色器，将 Albedo 颜色改为绿色。此材质的 Inspector 设置如图 14-8 所示。

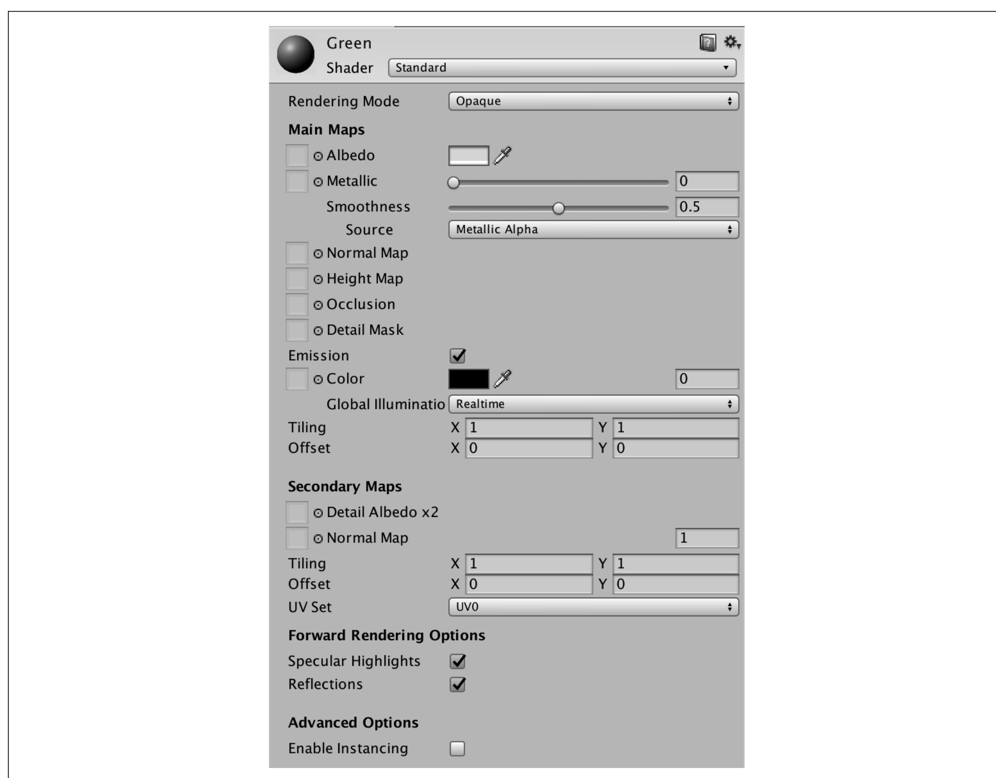


图 14-8：Green 材质的设置

接下来创建世界中的对象。

(3) 打开 GameObject 菜单，选择 3D Object → Cube，**创建一个立方体**。将该立方体命名为 floor，位置设为 (0, 0, 0)，缩放设为 (10, 1, 10)。

(4) **再创建一个立方体**，命名为 Wall 1，位置设为 (−1, 3, 0)，旋转设为 (0, 45, 0)。另外，将其缩放设为 (1, 5, 4)。

(5) **创建第三个立方体**，命名为 Wall 2，位置设为 (2, 3, 0)，旋转设为 (0, 45, 0)，缩放设为 (1, 5, 4)。

(6) **将 Green 材质拖放到 Wall 1 上**。

现在场景应如图 14-9 所示。

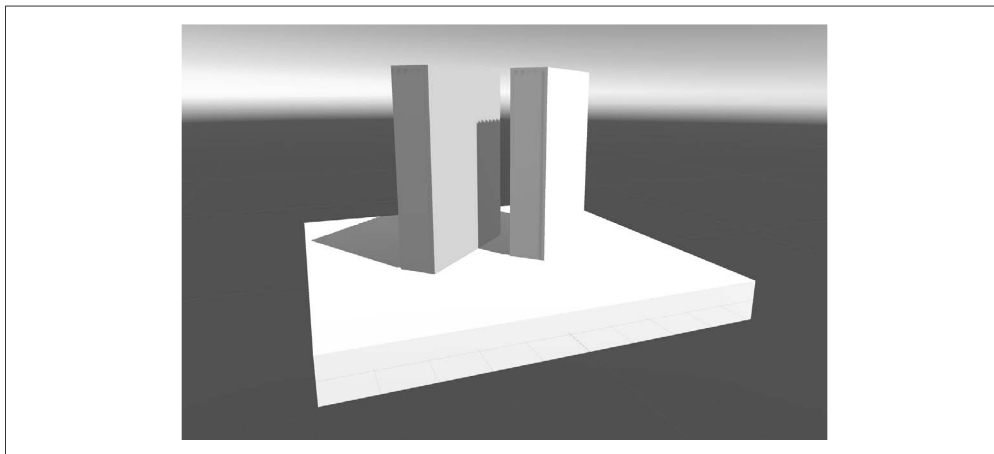


图 14-9：未使用光照贴图的场景（另见彩插）

我们将使 Unity 计算光照。

(7) **选择全部 3 个对象**（包括地面和两个墙壁），然后选中 Inspector 右上角的 Static 复选框（如图 14-10 所示）。

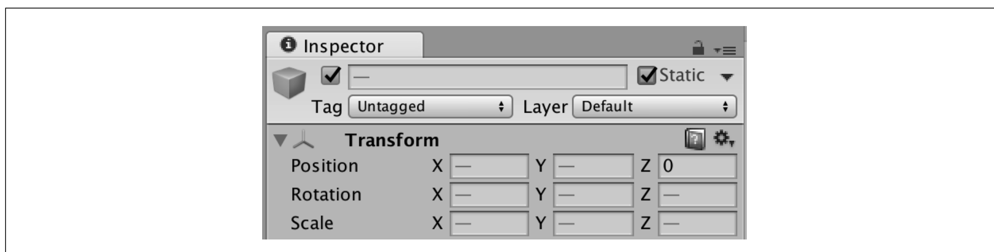


图 14-10：将对象设置为静态的

场景中有了静态对象后，Unity 将立即开始计算光照信息。过一会儿之后，光照将悄悄发生改变。最明显的效果是，绿色墙壁将把一些光线反射到白色墙壁上。比较图 14-11 与图 14-12 的不同之处。

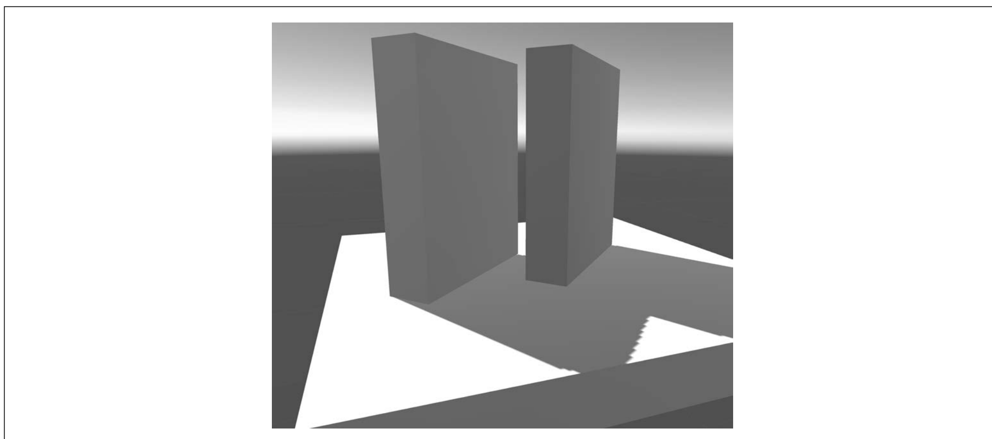


图 14-11：未激活全局光照的场景（另见彩插）

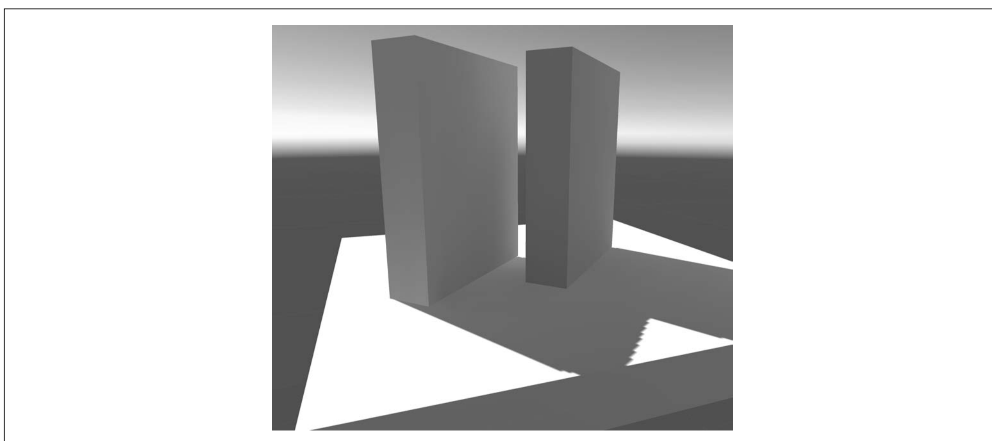


图 14-12：激活全局光照的场景，注意白色墙壁上的绿色反光（另见彩插）

这将使光照采用实时的全局光照。效果看上去很好，但是对性能会产生不小的冲击，因为只有部分光照计算是提前完成的。为了提升性能，但以更高的内存使用为代价，可以把光照**烘焙**（bake）成光照贴图。

(8) 选择 Directional Light，将 Baking 设置为 Baked。

稍候片刻，将计算光照，并把结果存储到一个光照贴图中。

全局光照对于静态对象的效果很好，但是对非静态对象没有效果。为了改进这一点，可以使用灯光探测器。

灯光探测器

灯光探测器是一个不可见对象，获取并记录来自各个方向的光照信息。邻近的非静态对象

可以使用此光照信息来照亮自己。

灯光探测器不是孤立工作的，而是成组创建的。它们在运行时，需要光照信息的对象根据探测器距离自己的远近，将最接近自己的几个探测器组合起来。例如，当对象接近一个反光的表面时，这样可以让对象自身反射更多光线。

接下来，我们在场景中添加一个非静态对象，然后添加一些灯光探测器，看看它们如何影响光照。

(1) 选择 GameObject → 3D Object → Capsule，**在场景中添加一个新胶囊体**。将胶囊体放在靠近绿色墙壁的位置。

注意，胶囊体并没有呈现从墙壁上反射过来的绿色光线。事实上，它接受了墙壁方向的**太多**光线，因为来自天空的光线是在该方向上照亮它的。墙壁应该阻挡这些光线。从图 14-13 可以看到这种效果。

(2) 打开 GameObject 菜单，选择 Light → Light Probe Group，**添加一些灯光探测器**。

一个球体集合将会出现，每个球体代表一个灯光探测器，显示了在空间中的该点接受的光照。

(3) **重新排列探测器**，不让它们嵌入到场景中。也就是说，它们都在空间中浮动，而没有嵌入到地面或墙壁中。



通过选择灯光探测器并单击 Edit Light Probes 选项，然后选择单独的探测器并移动，可以调整组中单独探测器的位置。

完成之后，胶囊体将显示从墙壁上反射的绿色光。比较图 14-13 与图 14-14 的区别。

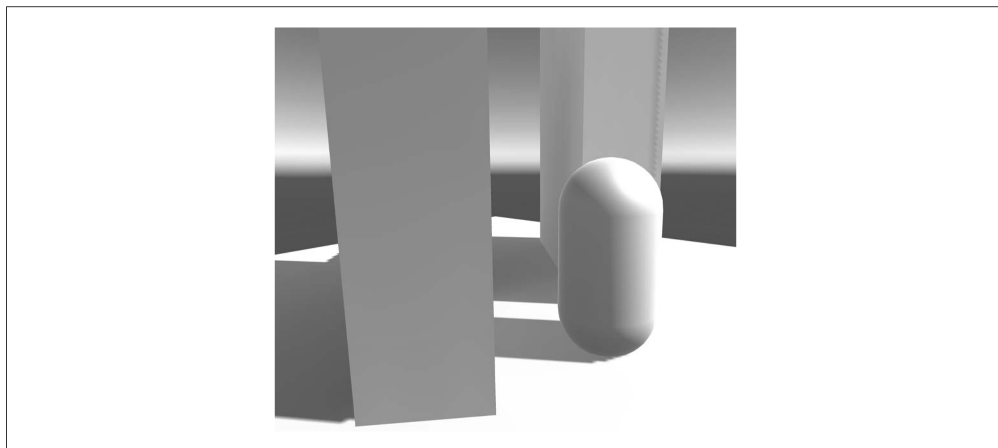


图 14-13：未使用灯光探测器的场景（另见彩插）

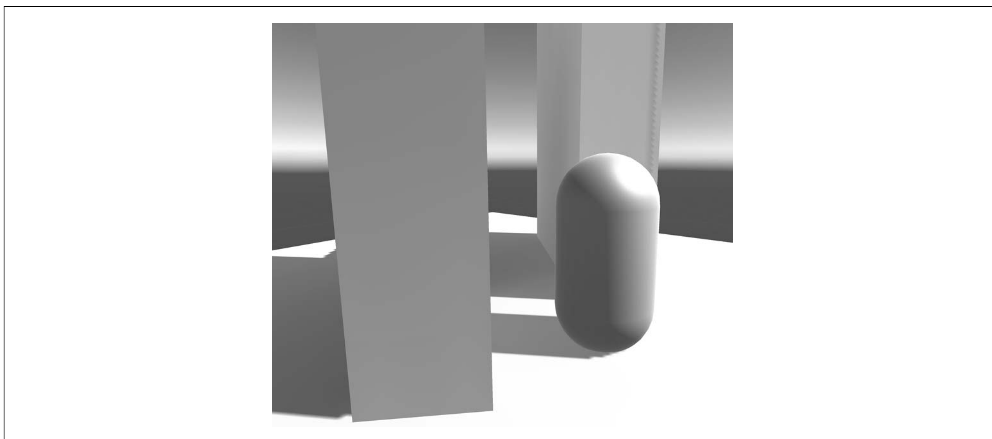


图 14-14：使用了灯光探测器的场景，注意绿色光线反射到了胶囊体上（另见彩插）



灯光探测器越多，光照的计算时间越长。另外，如果光照突然发生改变（即在不同区域之间），那么在靠近过渡区域的地方，应该更稠密地放置探测器，以免对象看上去格格不入。

14.3 性能考虑

最后，在结束本章的内容之前，我们来介绍 Unity 中内置的性能工具。游戏中使用的光照设置可能对用户的设备性能产生巨大的影响。另外，将对象标记为静态对象，除了能够实现全局光照和光照贴图，对性能也有影响。

不过，游戏的性能并不是完全由游戏的图形决定：脚本占用 CPU 的时间也可能产生巨大影响。

为了帮助处理性能问题，Unity 提供了几种工具和功能，可用来提升你的生产效率。

14.3.1 Profiler

Profiler 是一个工具，用来在玩游戏的过程中记录关于游戏的数据。它在每一帧中收集来自几个位置的信息，例如：

- 每一帧调用的脚本方法，以及调用它们所用的时间；
- 绘制一帧所需的“绘制调用”（即让图形芯片执行绘制工作的指令）的数量；
- 游戏使用的内存量，包括脚本和图形内存；
- 播放音频占用的 CPU 时间；
- 活动物理体的数量，以及每一帧中需要处理的物理碰撞的数量；
- 在网络上发送和接收的数据量。

Profiler 分为两个部分。上半部分分为若干行，每一行对应于每个数据记录器。图 14-15 显

示了一个 Profiler。在玩游戏时，每个记录器都将填充信息。下半部分显示了正在审查的具体帧的详细信息，这些信息来自当前选中的记录器。

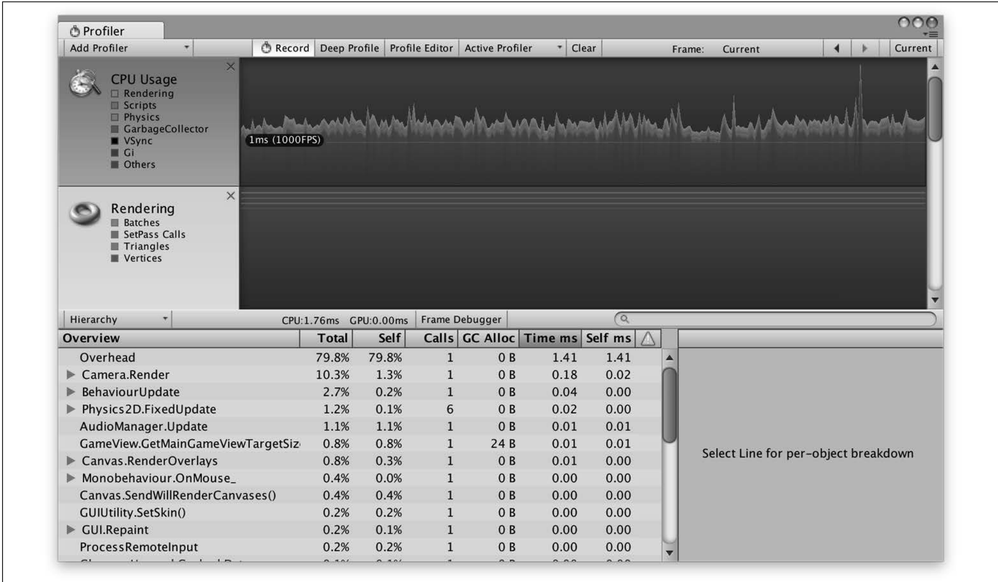


图 14-15：Profiler



在本书中看到的具体结果不一定与你在游戏中看到的完全相同。具体显示的数据取决于使用的硬件（包括运行 Unity 的计算机，以及用于测试游戏的移动设备），以及所使用的 Unity 具体版本。Unity Technologies 总是在修改场景背后的引擎，所以你很可能会看到不同的结果。

虽然如此，收集游戏性能数据的步骤是相同的，你可以把这些技术应用到几乎任何游戏上。

(1) **打开 Profiler，然后单击 Window 菜单，选择 Profiler**。或者，在 Mac 上按 Command-7 键，在 PC 上按 Ctrl-8 键。Profiler 将会显示。

要开始使用 Profiler，只需要让 Profiler 在游戏运行的过程中处于开启状态即可。

(2) 按 Play 按钮或 Ctrl-P（Mac 上为 Command-P）键**启动游戏**。

Profiler 将开始填充信息。如果游戏没有正在运行，那么分析游戏会简单许多。因此，在继续操作之前，需要让游戏退出 Play 模式。

(3) **过一会儿之后，停止或暂停游戏**。Profiler 中的数据不会消失。

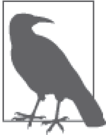
Profiler 已经停止填充数据，现在就可以仔细查看单独的帧了。

(4) **单击并拖动 Profiler 的顶行**，将会显示一条垂直线，Profiler 下半部分显示的数据将会更新，以反映选中的帧。

不同的记录器显示不同的信息。就 CPU 而言，默认显示 Hierarchy，即该帧中调用的全部方法的列表，并按照调用每个方法所需的时间进行排序（如图 14-16 所示）。也可以单击每一行左侧的三角形来打开它们，并查看该行调用的方法的信息。

Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
WaitForTargetFPS	67.1%	67.1%	1	0 B	2.72	2.72
Overhead	24.4%	24.4%	1	0 B	0.99	0.99
▶ Camera.Render	4.3%	0.4%	1	0 B	0.17	0.02
▶ BehaviourUpdate	1.0%	0.1%	1	0 B	0.04	0.00
Profiler.FinalizeAndSendFrame	0.4%	0.4%	1	0 B	0.01	0.01
GameView.GetMainGameViewTargetSiz	0.4%	0.4%	1	24 B	0.01	0.01
▶ Canvas.RenderOverlays	0.3%	0.1%	1	0 B	0.01	0.00
AudioManager.Update	0.2%	0.2%	1	0 B	0.01	0.01
▶ Monobehaviour.OnMouse_	0.1%	0.0%	1	0 B	0.00	0.00
Canvas.SendWillRenderCanvases()	0.1%	0.1%	1	0 B	0.00	0.00
GUIUtility.SetSkin()	0.1%	0.1%	1	0 B	0.00	0.00
▶ Physics2D.FixedUpdate	0.1%	0.0%	1	0 B	0.00	0.00

图 14-16：CPU Profiler 的 Hierarchy 视图



我们将花些时间着重介绍 CPU Profiler，因为理解它告诉我们的信息，有助于识别和修复游戏中可能发生的大量性能问题。

Hierarchy 的各列分别显示了每一行的不同信息。

Total

此列显示了在渲染这一帧时，调用该方法（以及该方法调用的方法）所用时间的百分比。

例如，在图 14-16 中，调用 `Camera.Render` 方法（Unity 引擎内部的一个方法）占用了渲染整帧所需时间的 4.3%。

Self

此列显示了当渲染这一帧时，调用该方法（且**仅限于**该方法）所用时间的百分比。这有助于识别究竟是什么导致了占用大量时间，是该方法自身，还是该方法所调用的方法。如果 Self 的值接近 Total 的值，则说明该方法自身占用了大量时间，而不是所调用的方法。

在图 14-16 中，`Camera.Render` 只占用了渲染这一帧所需时间的 0.4%，这说明该方法自身不会占用太多时间，但是它调用的方法会占用更多时间。

Calls

此列显示了在这一帧中调用该方法的次数。

在图 14-16 中，`Camera.Render` 只被调用了一次（很可能是因为场景中只有一个摄像机）。

GC Alloc

此列显示了在这一帧中，该方法必须分配的内存量。如果频繁分配内存，就增加了之后内存垃圾收集器必须运行的概率，进而导致延迟。

在图 14-16 中，调用 `GameView.GetMainGameViewTargetSize` 会分配 24 字节。虽然数字看起来不大，但是不要忘记，游戏在渲染尽可能多的帧。如果每一帧都分配了少量内存，那么随着时间增加，占用的内存会积聚起来，导致垃圾收集器必须进行清理，而这会损害游戏的性能。

Time ms

此列显示了调用该方法（以及该方法调用的所有方法）所需的时间量，单位为毫秒。在图 14-16 中，调用 `Camera.Render` 用了 0.17 毫秒。

Self ms

此列显示了调用该方法（且**仅限于**该方法）所需的时间量，单位为毫秒。在图 14-16 中，调用 `Camera.Render` 只用了 0.02 毫秒，另外的 0.15 毫秒用在了该方法调用的其他方法上。

Warnings

此列显示了 Profiler 发现的任何问题。Profiler 能够对自己记录的数据执行一些分析，并能够提供数量有限的建议。

14.3.2 获取设备数据

按照前一节的步骤操作时，收集到的数据来自 Unity Editor。但是，Editor 中游戏的性能特征与在设备上的不同。PC 或 Mac 一般具有比移动设备更快的 CPU、更多的 RAM 和更好的 GPU。因此，从 Profiler 得到的结果是不同的。针对 Editor 中游戏的性能数据进行优化，对最终用户而言可能并没有提升游戏性能。

为了解决这个问题，可以使用 Profiler 来收集在设备上运行的游戏数据。为此，执行下面的步骤。

(1) **按照后面 17.2 节的步骤，构建并在手机上安装游戏**。重要的是，确保开启 Development Build 和 Autoconnect Profiler。

(2) **确保设备和计算机在同一个 WiFi 网络中**，并且设备通过 USB 数据线连接到计算机。

(3) **在设备上启动游戏**。

(4) **打开 Profiler，然后打开 Active Profiler 菜单**。从显示的列表中选择你的设备。

Profiler 将开始从你的设备直接收集数据。

14.3.3 通用提示

下面这些技巧能够提升游戏的性能。

- 在 `Rendering` Profiler 中，尽量让 `Verts` 计数低于每一帧 200 000。
- 选择在游戏中使用的着色器时，从 Mobile 或 Unlit 分类中选择。相比于其他着色器，这些着色器更简单，而且在每一帧中需要的运行时间更少。
- 让场景中使用的不同材质数量尽可能少。另外，让尽可能多的对象使用**相同**的材质。这会让 Unity 更容易同时绘制这些对象，从而提升性能。

- 如果某个对象在场景中从来不会移动、改变大小或旋转，那么选中 Inspector 右上角的 Static 复选框。这将使引擎执行一些内部优化。
- 减少场景中的灯光数。灯光越多，引擎要做的工作就越多。
- 使用烘焙光照比实时光照更加高效。但是要记住，烘焙灯光不能移动，而且烘焙灯光信息会占用内存。
- 尽可能使用压缩的纹理，而不是未压缩的。压缩的纹理占用内存更少，并且引擎访问这些纹理需要的时间也更少（因为要读取的数据更少）。

还有许多有用的性能提示，参见 Unity 手册（https://docs.unity3d.com/Manual/OptimizingGraphicsPerformance.html）。

14.4　小结

光照能够使场景的效果看起来好很多。即使不打算让游戏看起来完全真实，做些工作来设置场景的光照，也能让整个游戏给玩家更好的感受。

关注游戏的性能也很重要。使用 Profiler 能够仔细查看游戏的实际表现，并利用这些信息来调整游戏。

第15章

在Unity中创建GUI

游戏是软件，所有的软件都要有用户界面。即使简单到启动新游戏的一个按钮，或者显示玩家当前得分的标签，游戏也需要一种方式来显示平凡的非游戏内容，以供玩家交互。

好消息是，Unity 有一个很棒的 UI 系统。Unity 4.6 引入了 UI 系统，这个系统极其灵活而强大，是针对游戏通常遇到的情景而设计的。例如，该 UI 系统支持 PC、游戏机和移动平台；允许一个 UI 适用多种屏幕大小；能够响应来自键盘、鼠标、触摸屏和游戏手柄的输入；支持在屏幕空间和世界空间显示 UI。

简言之，这是一个令人难以置信的工具包。我们在第二部分和第三部分的游戏中构建了 GUI，但本章会介绍 GUI 系统的一些细微之处，使你能够充分利用其中的功能。

15.1 Unity GUI系统的工作方式

从根本上讲，Unity 中的 GUI 与场景中的其他可见对象并没有太大的区别。GUI 是应用了纹理的网格，由 Unity 在运行时构造。另外，GUI 包含脚本，能够响应鼠标移动、键盘事件和触摸，并更新和修改网格。网格是通过摄像机来显示的。

Unity 的 GUI 系统包含几个协同工作的组件，GUI 系统核心由几个具有 `RectTransform` 的对象构成，它们均包含在 Canvas 内，负责绘制内容并响应**事件**。

15.1.1 Canvas

Canvas（**画布**）是一个对象，负责将所有 UI 元素绘制到屏幕上。因此，屏幕也是绘制画布的全部空间。

所有 UI 元素都是 Canvas 的子对象——如果按钮不是 Canvas 的子对象，就无法显示。

Canvas 用来决定 UI 的绘制**方式**。另外，通过附加一个 Canvas Scaler 组件，可以控制 UI

元素的缩放。15.5 节将详细介绍 Canvas Scaler。

Canvas 可用在 3 种模式下：Screen Space - Overlay、Screen Space - Camera 和 World Space。

- 在 Screen Space - Overlay 模式下，整个 Canvas 将在游戏之上绘制。也就是说，场景中的所有 Camera 将它们看到的游戏内容绘制到屏幕上，然后 Canvas 将被绘制到这些内容之上。这是 Canvas 的默认模式。
- 在 Screen Space - Camera 模式下，Canvas 的内容被渲染到一个平面上，该平面被置于 3D 空间中，位于指定的 Camera 前方一定距离处。当 Camera 移动时，Canvas 的位置也随之改变，以保持在相对于 Camera 的相同点上。在这种模式下使用时，Canvas 实际上是一个 3D 对象，这就意味着处于 Canvas 与 Camera 之间的对象将遮挡 Canvas。
- 在 World Space 模式中，Canvas 是场景中的一个 3D 对象，具有自己的位置和旋转，独立于场景中的任何 Camera。这意味着你可以创建一个 Canvas，例如，使其包含一个开门用的密码键盘，然后将其置于门的旁边。



如果你曾经玩过游戏《毁灭战士》(2016) 或《杀出重围：人类革命》，那么你已经与世界空间 GUI 交互过了。在这些游戏中，玩家会与游戏内的计算机屏幕交互。他们走到计算机屏幕旁边，并“点击”屏幕上显示的按钮。

15.1.2 RectTransform

Unity 是一个 3D 引擎，这意味着所有对象都有一个 Transform 组件，决定其在 3D 空间中的位置、旋转和缩放比例。然而，Unity 中的 GUI 是 2D 的。这意味着其中所有 UI 元素都是 2D 矩形，具有位置、宽度和高度。

UI 对象有一个 `RectTransform` 对象，用于控制这种设置。`RectTransform` 代表一个矩形，UI 内容可在这个矩形内显示。重要的是，如果一个 `RectTransform` 是**另一个** `RectTransform` 的子对象，那么子对象的位置和大小可相对于父对象来设置。

例如，Canvas 对象有一个 `RectTransform`，至少定义了 GUI 的大小。另外，构成游戏 GUI 的全部 GUI 元素都有自己的 `RectTransform`。因为这些 GUI 元素是 Canvas 的子对象，所以其 `RectTransform` 的位置将**相对于** Canvas 来确定。



Canvas 的 `RectTransform` 还可以定义 GUI 的位置，但是这取决于 Canvas 是在**屏幕空间**(screen-space)、**摄像机空间**(camera-space) 还是**世界空间**(world-space) 内。如果 Canvas 不在**世界空间**内，那么其位置将被自动确定。

当嵌套多个子对象时，这一点也适用。如果创建一个有 `RectTransform` 的对象，然后添加其子对象（每个子对象都有自己的 `RectTransform`），那么这些子对象的位置将相对于它们的父对象来确定。



`RectTransform` 并不局限于 UI 元素。你可以为任何对象添加 `RectTransform`。此时，`RectTransform` 将取代 Inspector 顶部的 `Transform` 组件。

15.1.3 Rect工具

Rect 工具提供了一种简单的方法，来移动具有 `RectTransform` 组件的对象，以及调整这些对象的大小。可按 T 键激活 Rect 工具，或者从 Unity 窗口左上角的工具栏选择 Rect 工具（如图 15-1 所示）。



图 15-1：在工具栏中选择 Rect 工具

启用 Rect 工具后，选定对象周围将出现一组手柄围绕成的矩形（如图 15-2 所示）。拖动这些手柄时，对象的大小和位置将发生改变。

另外，如果将鼠标光标移动到手柄附近，但位于矩形之外，光标将发生改变，提示进入旋转模式。此时单击并拖动，对象将绕其轴点旋转，轴点即对象中间的圆圈。如果选定对象具有 `RectTransform` 组件，可以通过单击拖动来移动轴点。

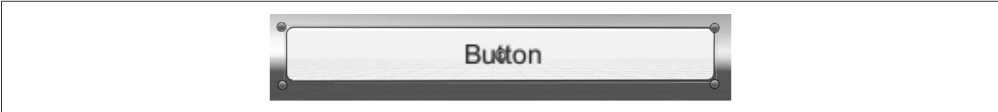
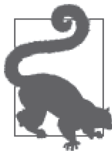


图 15-2：Rect 工具的手柄，轴点位于中心



Rect 工具并不局限于 UI 元素，也可以用于 3D 对象。选中一个 3D 对象时，Rect 工具会根据场景视图中查看对象的方式，确定矩形和手柄的位置。图 15-3 给出了一个例子。

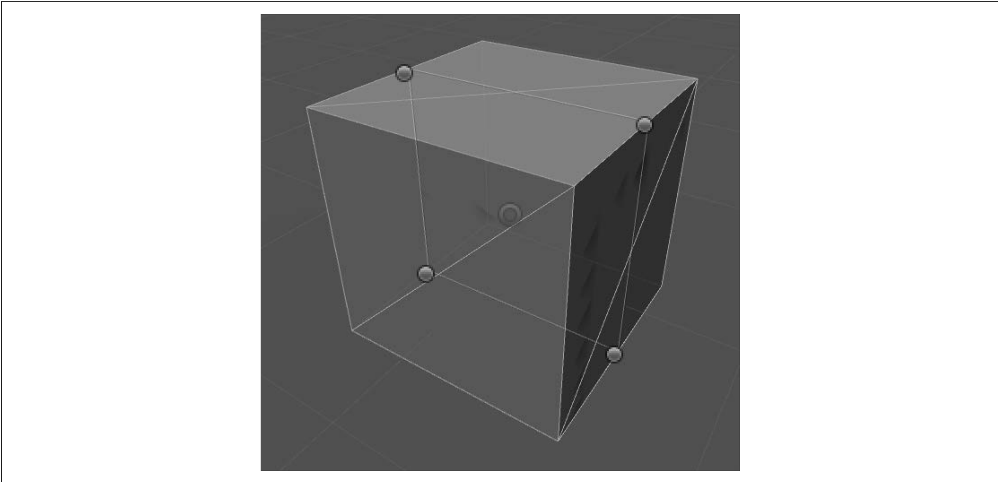


图 15-3：Rect 工具的手柄，围绕一个 3D 立方体

15.1.4 锚点

当一个 `RectTransform` 是另一个 `RectTransform` 的子组件时，其位置将相对于父组件的**锚点**来确定。这就允许我们定义父矩形的大小与子矩形的位置和大小之间的关系。例如，可以将一个矩形定位到其父矩形的底部，并填满父矩形的宽度；当父矩形的大小变化时，子矩形的位置和大小也将被更新。

在 `RectTransform` 的 Inspector 中，可以看到一个允许选择锚点预设值的框（如图 15-4 所示）。

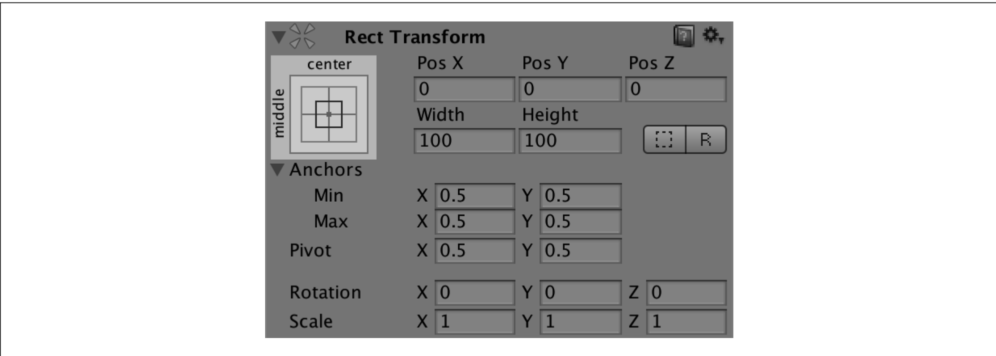


图 15-4：显示了当前为 `RectTransform` 的锚点选定的预设值的框

单击此框将显示一个小弹出窗口，允许修改预设值（如图 15-5 所示）。

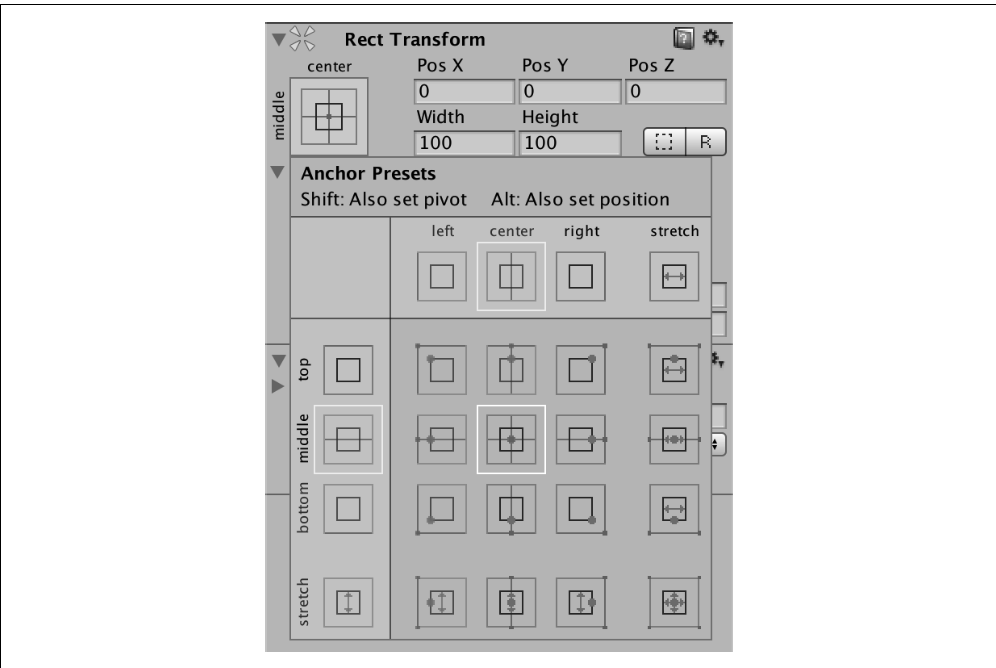


图 15-5：锚点预设值选择面板

单击其中任意一个预设值，将修改 `RectTransform` 的锚点。这不会修改矩形的位置和大小，修改的是当其**父矩形**的大小改变时，矩形的大小随之改变的方式。



这是 GUI 系统中的一个非常偏重视觉感受的部分，因此了解其工作方式最好的方法是自己多尝试。把一个 Image 游戏对象放到另一个 Image 游戏对象内，然后试着修改子视图的锚点，再调整父视图的大小。

15.2 控件

在场景中，有几种控件可供使用。其中既包括随处可见的简单控件，如按钮和文本字段，也包括复杂的控件，如滚动视图。

本节将讨论最重要的一些控件，以及如何使用它们。因为会有新的控件陆续添加进来，所以如果想查看完整的控件列表，请参考 Unity 手册。



Unity GUI 系统中的控件常常由多个游戏对象构成，它们彼此协同工作。因此，当你在画布中添加了一个控件，结果发现 Hierarchy 中多了好几个对象时，也不必惊讶。

15.3 事件和光线投射

当用户触摸屏幕上的按钮时，他们期望按钮执行为其配置的任务。为此，UI 系统必须能够知道触摸了**哪个**对象。

支持这种功能的系统称为**事件系统**。事件系统很复杂：除了为 GUI 提供输入，还能够作为一种通用解决方案，用于识别游戏中的对象何时被单击、触摸或拖动。



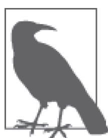
事件系统由 EventSystem 对象表示，创建 Canvas 时该对象就会出现。

事件系统基于**光线投射**（raycast）原理。在光线投射中，用户触摸屏幕的点会发出一条**光线**（ray）——不可见的线。这条光线一直前进，直到碰到某个东西，此时事件系统就会知道用户手指“下方”是什么对象。

因为光线投射系统在 3D 空间中工作，就像引擎的其余部分一样，所以事件系统既能够处理 2D GUI，也能够处理 3D GUI。当发生一个事件时，例如手指触摸或鼠标单击，场景中的每个**光线投射器**都会发射光线。光线投射碰撞器有 3 种不同的类型：**图形光线投射器**、**2D 物理光线投射器**和 **3D 物理光线投射器**。每一种类型都查看光线可能碰到的不同对象。

- 图形光线投射器检查自己的光线是否与画布内的任何 Image 组件发生碰撞。
- 2D 物理光线投射器检查自己的光线是否与场景中的任何 2D 碰撞器发生碰撞。
- 3D 物理光线投射器检查自己的光线是否与场景中的任何 3D 碰撞器发生碰撞。

当用户触摸按钮 GUI 时，附加到 Canvas 的图形光线投射器组件会从手指接触屏幕的位置发射一条光线，并检查这条光线是否碰撞到了任何 Image。因为按钮有一个 Image 组件，所以光线投射器会向事件系统报告称触摸了按钮。



2D 和 3D 物理光线投射器不在 GUI 系统中使用，但可以用来检测单击、触摸和拖动场景中的 2D 和 3D 对象的操作。例如，可以使用 3D 物理光线投射器检测用户什么时候单击了一个立方体。

响应事件

构建自定义 UI 时，为自己的 UI 元素添加自定义行为通常极其有用。一般来说，这需要能够获得关于输入事件（如单击和拖动）的通知。

为了创建能够响应输入事件的脚本，需要使类符合特定的接口，然后实现这些接口要求的方法。例如，`IPointerClickHandler` 接口要求实现本接口的类具有一个签名如下所示的方法：`public void OnPointerClick (PointerEventData eventData)`。当事件系统检测到当前触控点（可能是鼠标光标或者触摸屏幕的手指）执行"单击"操作时，即在图片边界内，鼠标按键按下并松开，或者手指按下并抬起，就会调用此方法。

为了进行演示，下面提供了一个小教程来说明如何响应 GUI 对象内的触控点单击。

(1) **在一个空场景中**，打开 GameObject 菜单，选择 UI → Canvas，**创建一个新的 Canvas**。场景中将添加一个 Canvas。

(2) 打开 GameObject 菜单，选择 UI → Image，**创建一幅新图片**。这将把一个 Image 对象作为 Canvas 的子对象添加进来。

(3) **为 Image 对象添加一个新的 C# 脚本**，命名为 EventResponder.cs，并在文件中添加下面的代码：

```
//对于访问IPointerClickHandler和PointerEventData而言必不可少
using UnityEngine.EventSystems;

public class EventResponder : MonoBehaviour,
  IPointerClickHandler {

    public void OnPointerClick (PointerEventData eventData)
    {
        Debug.Log("Clicked!");
    }

}
```

(4) **运行游戏**。单击图片时，控制台将显示单词"Clicked!"。

15.4 使用布局系统

创建一个新的 UI 元素时，通常把元素直接添加到场景中，然后手动设置其大小和位置。但是，在以下两种情况中，这很快会变得难以维护：

- 因为游戏将在不同尺寸的屏幕上显示，所以无法知道画布的大小时；
- 当需要在运行时添加和删除 UI 中的内容时。

在这些情况中，可以利用 Unity GUI 系统内置的布局系统。

为了说明其工作方式，我们快速创建一个纵向的按钮列表。

(1) 在 Hierarchy 中单击**选择一个 Canvas 对象**。（如果还没有 Canvas 对象，就打开 GameObject 菜单，选择 UI → Canvas 来创建一个。）
(2) 打开 GameObject 菜单并选择 Create Empty Child，或者按 Ctrl-Alt-N（Mac 上为 Command-Option-N）键，**创建一个新的空子对象**。
(3) **将新对象命名为 List**。
(4) 打开 GameObject 菜单，然后选择 UI → Button，**创建一个新 Button**。设新 Button 为 List 对象的子对象。
(5) **为 List 对象添加一个 Vertical Layout Group 组件**。选择 List 对象，单击 Add Component 按钮，然后选择 Layout → Vertical Layout Group（也可以输入 vertical layout group 的前几个字母来快速选择此对象）。

你会注意到，当把 Vertical Layout Group 添加到 List 对象时，Button 的大小将发生变化，填满 List 对象的整个矩形。图 15-6 和图 15-7 分别显示了添加该组件之前和之后的情形。

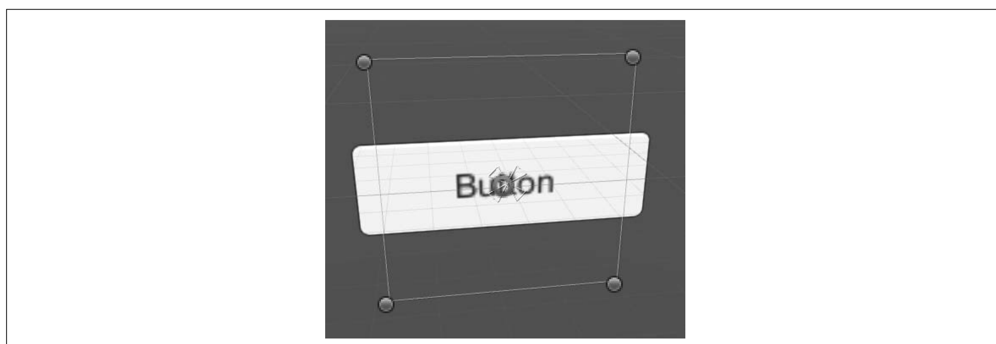


图 15-6：把 Vertical Layout Group 添加到 List 对象之前的按钮

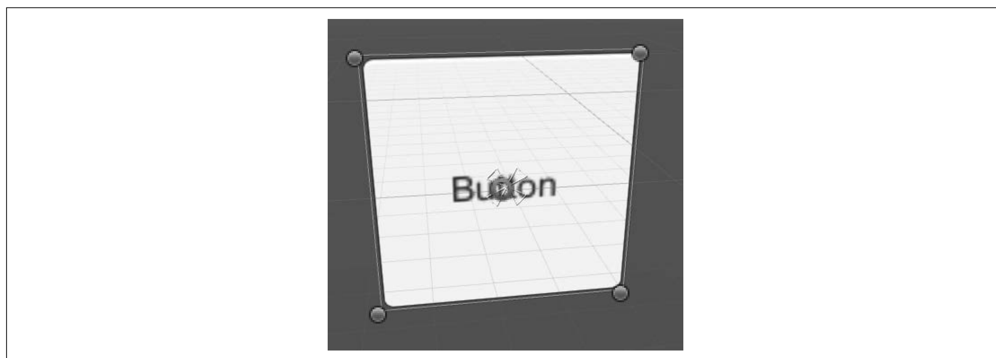


图 15-7：把 Vertical Layout Group 添加到 List 对象之后的按钮

接下来，注意当布局组中包含**多个**按钮时会发生什么。

(6) **选择 Button**，按 Ctrl-D（Mac 上为 Command-D）键**复制该对象**。

执行此操作时，原按钮和副本按钮的位置和大小将立即发生改变，使它们都能适合 List 对象（如图 15-8 所示）。

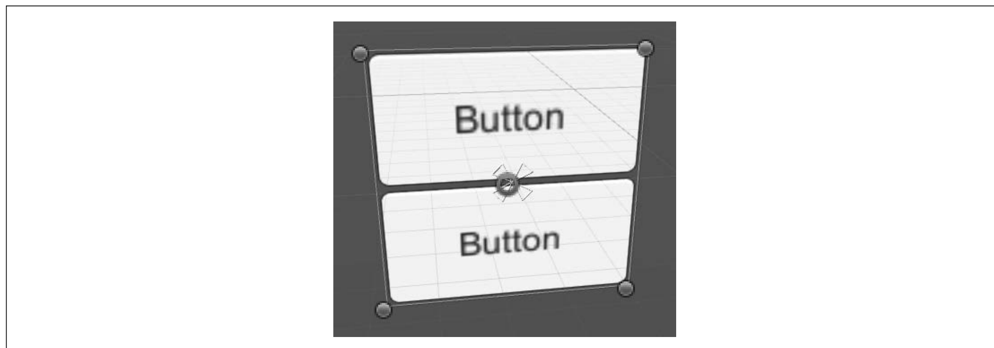


图 15-8：垂直布局的两个按钮

除了 Vertical Layout Group，GUI 系统还包含 Horizontal Layout Group，其工作方式与 Vertical Layout 完全相同，只不过是横向排列。另外，还有一个 Grid Layout Group，将内容布局到一个规则网格中，允许显示多行内容，这些内容会在必要时进行换行。

15.5 缩放Canvas

你的游戏可能在不同类型的屏幕上显示，这些屏幕不仅尺寸可能不同，**显示密度**（display density）也可能不同。显示密度指的是单个像素的大小。在现代移动设备上，屏幕的显示密度通常比较高。

视网膜显示屏是一个比较引人注目的例子。iPhone 4 以后的所有 iPhone，以及 iPad 3 以后的所有 iPad 都采用了视网膜显示屏。这些设备的屏幕尺寸与之前的模型相同，但是显示密度增加了一倍。例如，iPhone 3GS 的屏幕宽度为 320 像素，而 iPhone 4 的屏幕宽度为 640 像素。屏幕上显示的内容物理尺寸保持不变，而显示密度的增加则意味着显示效果更加平滑、更加美观。

由于 Unity 处理的是单独像素，在更高密度的屏幕上渲染 GUI，将导致 GUI 内容显示为原大小的一半。

为了解决这个问题，Unity GUI 系统包含了一个 Canvas Scaler 组件。Canvas Scaler 的作用是自动调整所有 GUI 元素的比例，确保在当前运行游戏的屏幕上，这些元素显示为合适的大小。

通过 GameObject 菜单创建 Canvas 对象时，将自动添加一个 Canvas Scaler 组件。Canvas Scaler 能够在 3 种模式下工作：Constant Pixel Size、Scale With Screen Size 和 Constant Physical Size。

Constant Pixel Size

默认模式。在此模式下，Canvas 不会基于屏幕尺寸或密度而缩放。

Scale With Screen Size

此模式使 Canvas 根据自己的大小，以及相对于“参照分辨率”（在 Inspector 中指定）的比值来缩放内容。例如，如果将参照分辨率设为 640 × 480，那么在分辨率为 1280 × 960 的设备上玩游戏时，每个 UI 元素将放大一倍。

Constant Physical Size

此模式使 Canvas 根据游戏设备报告的 DPI（每英寸[1]点数）缩放内容，前提是此值可用。



根据我们的经验，在大部分情况下，Scale With Screen Size 是最有用的模式。

15.6 画面切换

大部分游戏 GUI 可分为两种类型：菜单和游戏内 GUI。菜单 GUI 是玩家为准备开始游戏而交互的，也就是说，玩家可选择开启新游戏或者继续之前的游戏，可选择配置设置，还可以搜索并加入多人游戏。游戏内 GUI 则被叠加到玩家所见的游戏世界之上。

游戏内 GUI 的结构一般不太会改变，并且通常会显示一些重要信息：玩家的箭筒中还有多少支箭，生命值还剩多少，到下一个目标的距离有多远。与之相反，菜单的结构通常变化较大。由于不同的结构要求，主菜单的外观通常与设置画面有显著的区别。

因为 GUI 只不过是摄像机渲染的一个对象，所以 Unity 实际上并没有内容“画面”的概念。它看到的只不过是画布中当前存在对象的集合。想要从一个画面移动到另一个画面，有两种选择：改变摄像机当前观看的画布；移动摄像机，使其观看不同的内容。

如果想要改变 GUI 元素的子集，那么改变画布的方法很合适。例如，如果你想让大部分装饰性 GUI 元素可见，但是改变 GUI 的某个部分，那么更合理的做法是修改画布，而不是调整摄像机。但是，如果想完全替换 GUI 元素，那么调整摄像机的位置会更高效。

需要记住一点：如果想让摄像机能够独立于画布移动，必须把画布模式设为 World Space；在 Screen Space-Overlay 和 Screen Space-Camera 模式下，UI 始终出现在摄像机正前方。

15.7 小结

我们已经看到，Unity 的 GUI 系统庞大且强大。你可以在不同的场景中，以不同的方式使用这个 GUI 系统。此外，其灵活的设计让你能够准确构建自己需要的 GUI。

需要特别记住的是，游戏的 UI 是最重要的组件之一。用户通过 UI 与游戏交互，在移动设备上，UI 是游戏控件的根本。你应该准备好在优化 UI 上投入大量时间。

注 1：1 英寸约合 2.54cm。

第16章

编辑器扩展

在 Unity 中构建游戏意味着处理大量游戏对象，以及构成这些游戏对象的组件。Unity 中的 Inspector 已经处理了许多工作，它将脚本中的所有变量自动展开为易于使用的文本字段、复选框以及可以拖放资源和场景对象的框，使构建场景的过程加快了许多。

然而，有时仅使用 Inspector 还不能满足我们的要求。Unity 的设计目标是尽可能简化 2D 和 3D 环境的构建过程，但是 Unity 的开发人员不可能预见游戏中可能出现的所有内容。

自定义编辑器允许你控制编辑器本身。自定义编辑器既可以是很小的增件窗口，允许自动执行编辑器内的常见任务，也可以复杂到完全覆盖 Unity 的 Inspector。

当创建的游戏比本书构建的游戏更加复杂时，我们发现，编写自己的工具来自动完成重复性任务能够节省大量时间。这并不是说，作为游戏开发人员，你的主要任务应该是编写软件来帮助构建游戏。你的主要任务应该是创建游戏！但是，如果你发现自己要实现的东西是重复性的，或者用现有的 Unity 功能很难实现，就可以考虑编写编辑器扩展来完成这个任务。



本章在一定程度上进入了 Unity 的后台。事实上，我们将使用 Unity 编辑器自身使用的类和代码。因此，本章的代码比前面编写的代码可能更加复杂难懂一些。

扩展 Unity 有若干种方式。本章将介绍其中的 4 种方法，每一种都比前一种更复杂、更强大一些，列举如下。

- 自定义向导提供了一种简单的方法来获取输入，以及在场景中执行一些操作。
- 自定义编辑器窗口允许创建自己的窗口和选项卡，其中可包含你需要的任意控件。

- 自定义属性绘制器允许在 Inspector 中为自己的数据类型创建自定义用户界面。
- 自定义编辑器允许完全覆盖对象的 Inspector。

为了学习本章的示例，最好创建一个新项目。

(1) **创建一个新项目**，命名为 Editor Extensions。将其设为 3D 项目，然后选择一个位置保存该项目（如图 16-1 所示）。

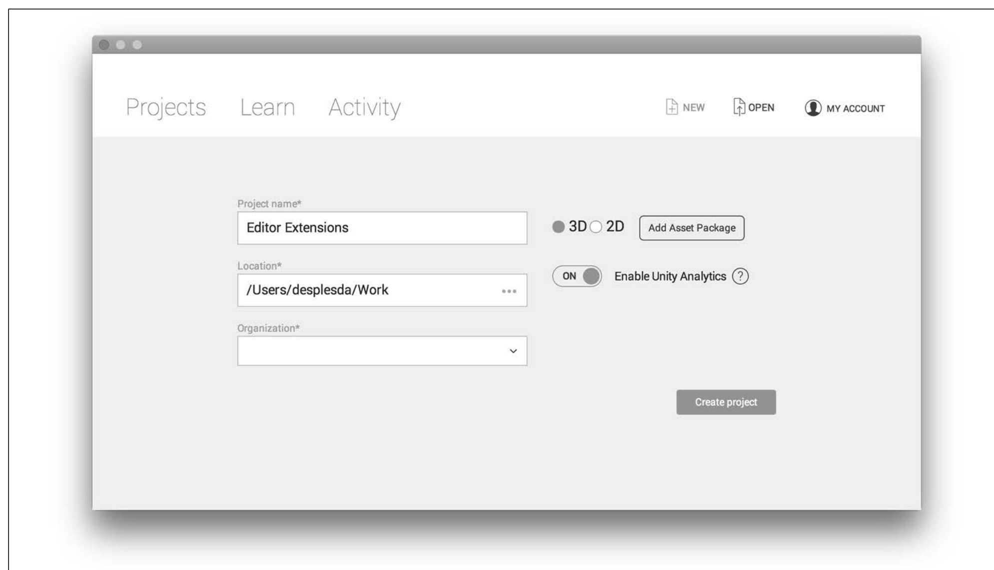


图 16-1：创建一个新项目

(2) **Unity 加载后，在 Assets 文件夹中创建一个新文件夹**。将新文件夹命名为 Editor。我们的编辑器扩展脚本将放到此文件夹中。

注意，将此文件夹命名为 Editor 非常重要，拼写和大小写都要正确。Unity 会专门查找具有此名称的文件夹。



此文件夹可放在任何位置，不一定是 Assets 文件夹的子文件夹，只要名称为 Editor 即可。这一点很有用，因为这意味着在较大的项目中，可以有多个 Editor 文件夹，处理具有大量脚本的情况就简单多了。

完成上述设置后，就可以开始创建自定义编辑器脚本了。

16.1 创建自定义向导

首先创建一个自定义向导。向导能够显示一个窗口，用来获得用户输入，然后可使用获得的输入在场景中执行一些操作。一个常见的例子是，根据提供的设置，在场景中创建不同的对象。



向导与 16.2 节讨论的编辑器窗口在概念上有相似点，二者都显示一个包含控件的窗口。它们的区别在于构造方式：Unity 替你处理向导的控件，但是编辑器窗口的控件则完全由你负责。当完成某个任务并不需要特殊的 UI 时，适合使用向导；当需要控制显示的内容时，编辑器窗口更加合适。

想要理解在你日常使用 Unity 时向导有何帮助，实际创建一个向导是最好的方法。我们将创建一个向导，该向导创建游戏对象，用来在场景中显示四面体（即三角形金字塔，如图 16-2 所示）。

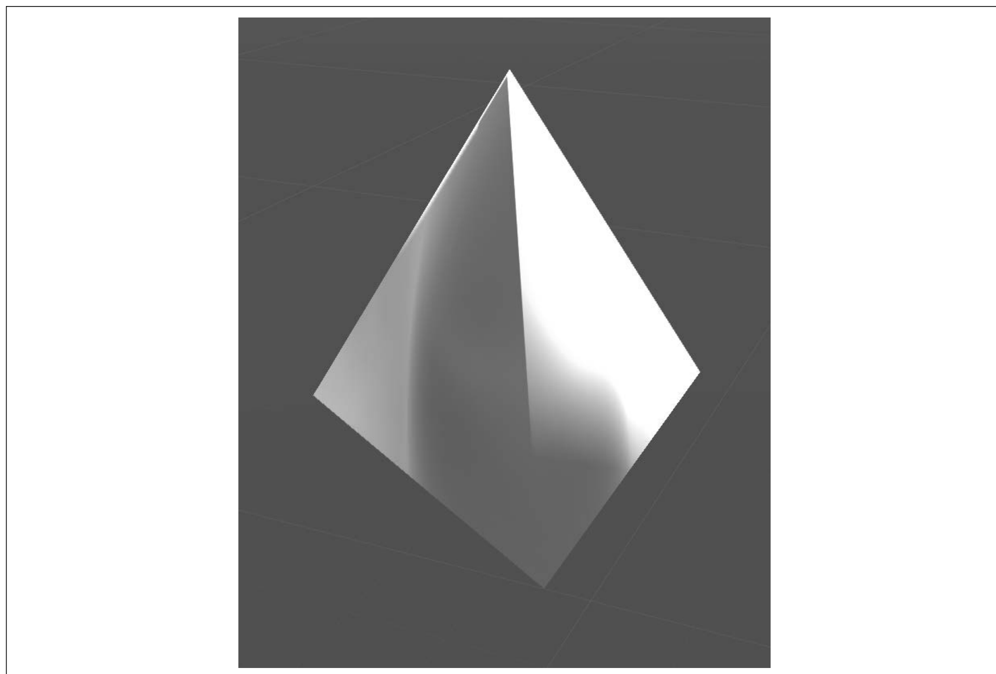


图 16-2：向导创建的四面体

创建这样的对象，需要先手动创建一个 `Mesh` 对象。通常这些对象是从文件中导入的，如第 9 章使用的 .blend 文件，不过用代码也可以创建这样的对象。

有了 `Mesh` 后，可以创建一个对象来渲染该网格。为此，首先创建一个新的 `GameObject`，然后附加两个组件：`MeshRenderer` 和 `Meshfilter`。然后，就可以在场景中使用该对象了。

自动执行这些步骤很简单，这意味着很适合为它们创建向导。

(1) 在 Editor 文件夹中**创建一个新的 C# 脚本**，命名为 Tetrahedron.cs，并添加下面的代码：

```
using UnityEditor;

public class Tetrahedron : ScriptableWizard {

}
```

ScriptableWizard 类定义了向导的基本行为。我们将实现一些方法来覆盖它，以实现我们需要的行为。

我们将实现一个方法来显示向导，涉及两项工作：首先，需要在 Unity 的菜单中添加一个菜单项，供用户调用该方法；其次，在此方法内，需要告诉 Unity 显示向导。

(2) **将下面的代码添加到 Tetrahedron 类中：**

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

    //可任意命名此方法，重要的是该方法是静态的，且具有MenuItem特性
    [MenuItem("GameObject/3D Object/Tetrahedron")]
    static void ShowWizard() {
      //第一个参数是标题，第二个参数是Create按钮上的标签
      ScriptableWizard.DisplayWizard<Tetrahedron>(
        "Create Tetrahedron", "Create");
    }
```

当附加到一个静态（static）方法时，MenuItem 特性使 Unity 在程序菜单中添加一个菜单项。本例中，GameObject → 3D Object 菜单中创建了一个名为 Tetrahedron 的新菜单项，选择此菜单项将调用 ShowWizard 方法。



实际上这个方法不一定要命名为 ShowWizard。你可以随意命名，Unity 只会查找 MenuItem 特性。

(3) **返回 Unity**，打开 GameObject 菜单。选择 3D Object → Tetrahedron，将出现一个空向导窗口（如图 16-3 所示）。

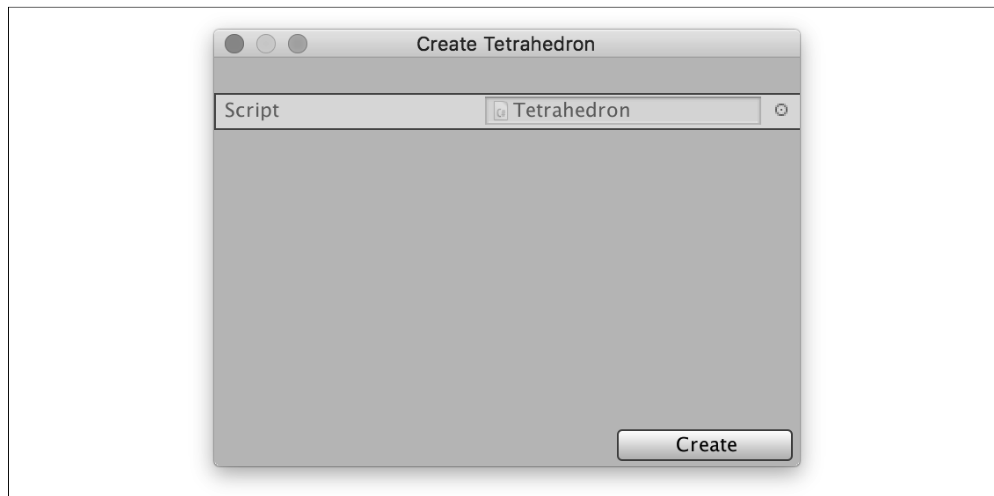


图 16-3：一个空向导窗口

接下来，在向导的类中添加一个变量。这将使 Unity 在向导窗口中为此变量显示合适的控件，就像在 Inspector 中一样。变量的类型为 Vector3，代表对象的高度、宽度和深度。

(4) **向 Tetrahedron 添加以下变量，代表四面体的大小：**

```
//此变量的显示效果就像在Inspector中一样
public Vector3 size = new Vector3(1,1,1);
```

(5) **返回 Unity**。关闭向导再重新打开，将看到 Size 变量的输入框（如图 16-4 所示）。

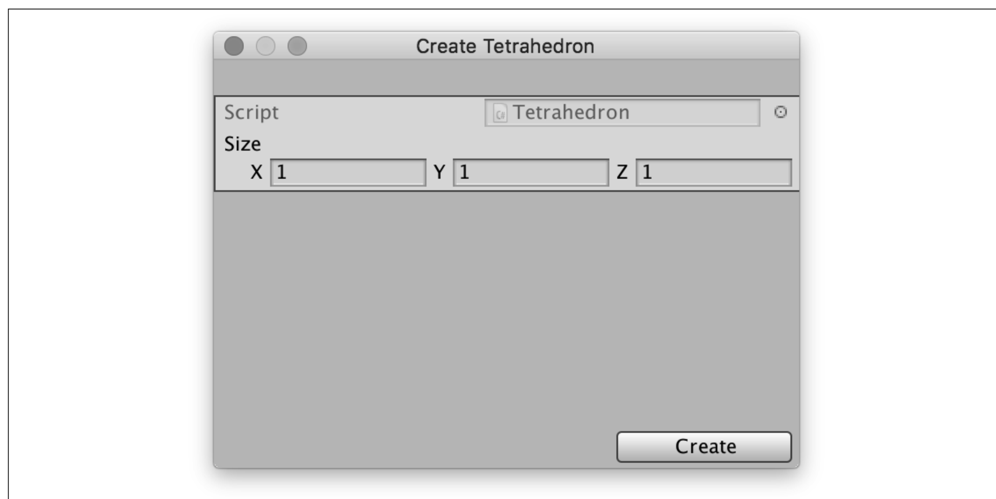


图 16-4：显示了 Size 变量控件的向导

现在，向导允许你向该变量提供数据，但是还不会使用该数据做任何事情。我们马上来解决这个问题。

调用 DisplayWizard 方法时，我们提供了两个字符串：第一个字符串是菜单名称，第二个字符串是向导的 Create 按钮中应该显示的文本。触摸此按钮时，向导的类将会收到对 OnWizardCreate 方法的调用，说明用户已经完成了对向导提供信息的操作。当 OnWizardCreate 方法返回后，Unity 将关闭窗口。

我们现在来实现 OnWizardCreate 方法，该方法将完成向导的大部分实际工作。它创建使用 Size 变量的 Mesh，并构造一个游戏对象来渲染该网格。

(6) **将以下方法添加到 Tetrahedron 类：**

```
//当用户单击Create按钮时调用
void OnWizardCreate() {

  //创建一个网格
  var mesh = new Mesh();

  //创建4个点
  Vector3 p0 = new Vector3(0,0,0);
  Vector3 p1 = new Vector3(1,0,0);
```

```
        Vector3 p2 = new Vector3(0.5f,
                                 0,
                                 Mathf.Sqrt(0.75f));
        Vector3 p3 = new Vector3(0.5f,
                                 Mathf.Sqrt(0.75f),
                                 Mathf.Sqrt(0.75f)/3);

        //根据size缩放
        p0.Scale(size);
        p1.Scale(size);
        p2.Scale(size);
        p3.Scale(size);

        //提供顶点列表
        mesh.vertices = new Vector3[] {p0,p1,p2,p3};

        //提供连接每个顶点的三角形列表
        mesh.triangles = new int[] {
          0,1,2,
          0,2,3,
          2,1,3,
          0,3,1
        };

        //使用此数据更新网格上的一些额外的数据
        mesh.RecalculateNormals();
        mesh.RecalculateBounds();

        //创建使用此网格的游戏对象
        var gameObject = new GameObject("Tetrahedron");
        var meshFilter = gameObject.AddComponent<MeshFilter>();
        meshFilter.mesh = mesh;

        var meshRenderer
          = gameObject.AddComponent<MeshRenderer>();
        meshRenderer.material
          = new Material(Shader.Find("Standard"));

    }
```

此方法首先创建一个新的 Mesh 对象，然后确定构成四面体的 4 个点的位置。接下来，根据 size 向量缩放这 4 个点，这意味着将重新确定它们的位置，使得到的四面体具有 size 的宽度、高度和深度。

这些点通过 Mesh 的 vertices 属性提供给该 Mesh，然后通过数字列表的方式来提供一个三角形列表。每个数字代表提供给 vertices 的一个点。

例如，在三角形列表中，0 指的是第一个点，1 指的是第二个点，以此类推。三角形列表使用由 3 个数组成的数字分组来定义三角形。例如，数字 0， 1， 2 表示网格将包含由 vertices 列表的第一个点、第二个点和第三个点构成的一个三角形。四面体由 4 个三角形构成：基底，以及 3 个侧面。因此，triangles 列表包含 4 个数字分组，每个分组由 3 个数字组成。

最后告诉网格，根据其包含的 vertices 和 triangles 数据，重新计算一些内部信息。然后，就可以在场景中使用它了：创建一个新的 GameObject，为其附加一个 MeshFilter，并提供跟我们刚才构建的 Mesh，然后附加一个 MeshRenderer 来实际显示 Mesh。最后，向 MeshRenderer 提供一个新的 Material，这是使用 Standard 着色器创建的，就像通过 GameObject 菜单创建的其他所有内置对象那样。

(7) **返回 Unity，关闭并重新打开向导窗口。**单击 Create 按钮时，场景中将添加一个新的四面体。如果修改 Size 变量，四面体的大小也将随之变化。

还要给向导添加最后一个功能。现在，向导并不检查 Size 变量的值是否合理。例如，向导应该拒绝创建一个高度为 −2 个单位的四面体。



严格来说，这么设置实际上是没有问题的，因为 Unity 能够处理这种情况。但是，知道如何进行这种输入验证会很有帮助。

对于本例而言，当 Size 变量的任意分量（*x*、*y* 或 *z*）值小于等于 0 时，我们将使向导拒绝创建四面体。

为此，我们将实现 OnWizardUpdate 方法。每当用户修改向导中的任意变量时，该方法就会被调用。通过使用该方法，我们有机会检查输入值，并相应地启用或禁用 Create 按钮。重要的是，我们可以添加说明文字，告诉用户**为什么**向导不接受输入。

(8) **在 Tetrahedron 类中添加以下方法：**

```
//每当用户在向导中做出任何修改时调用
void OnWizardUpdate() {

  //检查确保提供的值是有效的
  if (this.size.x <= 0 ||
    this.size.y <= 0 ||
    this.size.z <= 0) {

    //当isValid为true时，可单击Create按钮
    this.isValid = false;

    //解释为何如此
    this.errorString
      = "Size cannot be less than zero";
  } else {

    //由于用户可单击Create按钮，启用该按钮并清除任何错误消息
    this.errorString = null;
    this.isValid = true;
  }
}
```

将 isValid 属性设为 false 时，Create 按钮将被禁用，用户将无法单击该按钮。另外，如果将 errorString 属性设为 null 之外的值，窗口会显示一条错误消息，可以在消息中向用户解释问题所在（如图 16-5 所示）。

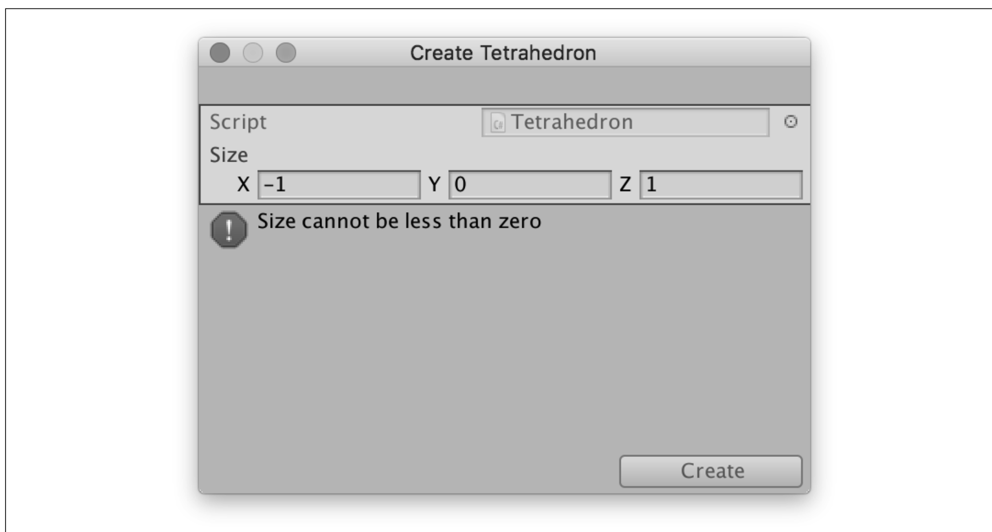


图 16-5：向导中显示错误消息

当执行重复工作时，或者仅凭 Unity 编辑器难以完成的工作时，向导能节省大量时间。编写向导很快，因为 Unity 编辑器会替你处理大部分用户界面。但是有些时候，你需要的功能是向导系统所不能提供的。我们接下来将介绍完全自定义的编辑器窗口。

16.2 创建自定义编辑器窗口

在 Unity 中，**窗口**指的是一块区域，可以是浮动的独立窗口，也可以是一个选项卡，停靠在 Unity 编辑器主界面的某个部分。



在 Unity 中，你所看到的几乎每个部分都是一个编辑器窗口。

创建编辑器窗口时，你能够完全控制其内容。这与向导和 Inspector 的工作方式不同，因为在向导和 Inspector 中，Unity 会替你自动绘制用户界面。在编辑器窗口中，除非你明确指定，否则什么也不会显示。这就为你提供了强大的能力，使你能够根据自己的需要，为 Unity 添加全新的功能。

本节将创建一个新编辑器窗口，统计项目中的纹理数。然而，在实现此功能之前，我们需要学习如何在编辑器窗口中进行绘制。

首先创建一个新的空编辑器窗口。

(1) **创建一个新的脚本**，命名为 TextureCounter.cs，将其放到 Editor 文件夹中。

(2) **打开新脚本**，用下面的代码替换文件内容：

```
using UnityEngine;
using System.Collections;
using UnityEditor;

public class TextureCounter : EditorWindow {

  [MenuItem("Window/Texture Counter")]
  public static void Init() {
    var window = EditorWindow
      .GetWindow<TextureCounter>("Texture Counter");
    //当加载一个新场景时，使此窗口不会被卸载
    DontDestroyOnLoad(window);
  }

  private void OnGUI() {
    //编辑器GUI放在这里
    EditorGUILayout.LabelField("Current selected size is "
      + sizes[selectedSizeIndex]);
  }

}
```

这段代码向 Window 菜单添加一个新菜单项，用于创建和显示一个使用 `TextureCounter` 类的新窗口。这段代码还标记出此窗口，告诉 Unity 当前场景改变时不应该卸载此窗口。

(3) **保存文件**，返回 Unity。

(4) **打开 Window 菜单**，将看到 Texture Counter 菜单项。单击该选项，将显示一个空窗口。

有了这个空窗口，就可以添加控件了。为此，我们需要知道如何使用 Editor GUI 系统。

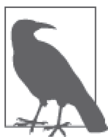
16.2.1 Editor GUI API

编辑器使用的 GUI 系统与用来构建游戏的 GUI 系统有很大区别。

在游戏的 GUI 系统（我们称之为 Unity GUI）中，我们创建游戏对象（代表文本标签、按钮等），并把它们置于场景中。

在用来创建编辑器 GUI 的 GUI 系统（我们称之为**立即模式** GUI，原因稍后介绍）中，我们调用特殊函数，使标签或按钮出现在特定的位置。每当 Unity 需要重绘屏幕的时候，就会调用这些函数。

术语“**立即模式**”（immediate mode）是指，调用这些特殊的 GUI 函数会使按钮立即显示在屏幕上，下一帧将清空屏幕，从屏幕上移除按钮和其他内容，然后再次调用 GUI 函数。这个过程将一直重复下去。



为了提高效率，Unity 并不会连续调用这些编辑器 GUI 函数。相反，它只在有可能需要时才会调用。例如，用户单击鼠标或按下按键时，包含 GUI 内容的窗口的大小改变时，以及其他与屏幕相关的事件发生时。

立即模式与 Unity GUI 系统的另外一个主要区别在于布局的工作方式。在 Unity GUI 中，对象的位置相对于其父对象及其锚点来确定。在立即模式 GUI 中，要么提供具体的矩形来描述想要绘制的东西的位置和大小，要么使用一个名为 `GUILayout` 的托管布局系统，我们稍后将介绍这个系统。

解释这种区别最好的方式是提供示例。接下来将详细介绍如何使用 GUI 系统，以及可以使用的不同控件。

1. Rect和布局

简单文本标签也许是能添加到窗口中的最简单的控件。我们将添加一些代码来完成此任务，然后再解释代码。

(1) **将下面的代码添加到 `OnGUI` 方法中：**

```
GUI.Label(                        ❶
        new Rect(50,50,100,20), ❷
        "This is a label!"      ❸
);
```

(2) **返回 Unity，打开编辑器窗口。**窗口中将显示文本“This is a label!”，如图 16-6 所示。



图 16-6：在编辑器窗口中手动添加的标签

下面来解释这段代码。

❶ 调用 `GUI` 类的 `Lable` 方法，作用是在窗口中显示文本。

❷ 创建一个新的 `Rect`，定义标签的 x 位置、y 位置、宽度和高度。在本例中，标签被放到了距离顶边和左边各 50 像素的地方，宽度为 100 像素，高度为 20 像素。

❸ 提供标签中实际显示的文本“This is a label!”。

每当 Unity 需要更新窗口中显示的内容时，就会运行这段代码。当调用 `GUI.Label` 方法时，就会把标签添加到窗口中。



对 GUI 函数的每个调用都必须发生在 OnGUI 方法内。如果在其他位置调用 GUI.Label 函数，那么会出现问题。

我们提供的 Rect 控制着标签出现的位置。对于本例这样简单的情形，这么做没有问题。但是对于更加复杂的情形，使用 Rect 就有些困难了。

为了帮助解决这个问题，立即模式 GUI 提供了一个方法，用来自动在垂直方向和水平方向上布局控件。

例如，为了以垂直堆叠列表的方式显示控件，可以创建一个新的 EditorGUILayout.VerticalScope，然后将其放到一个 using 语句中。

(3) **用下面的代码替换 OnGUI 方法的内容：**

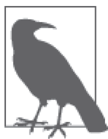
```
using (var verticalArea
  = new EditorGUILayout.VerticalScope()) {
          GUILayout.Label("These");
          GUILayout.Label("Labels");
          GUILayout.Label("Will be shown");
          GUILayout.Label("On top of each other");
}
```

本例中的标签有两个主要的不同点。

首先，注意调用的 Label 方法来自 GUILayout 类，而不是 GUI 类。这个版本的标签是在 VerticalScope 的上下文中调用的，因此能够正确定位自己的位置。

其次，不需要提供一个 Rect 来定义标签的位置和大小。它们将使用 VerticalScope 来确定这些信息。

这样的布局系统用起来更快，能够为程序员带来更好的体验。因此，本章剩余部分几乎全部会采用这个布局系统。



属性绘制器是例外，因为 GUI 布局系统对属性绘制器不起作用。在本节中，我们将回归手动方式布局控件，以及指定它们的矩形。

2. 控件如何工作

如前所述，立即模式 GUI 系统中的控件是一个函数调用。对于简单的控件，如标签，这很容易理解，但是对于允许用户提供输入的控件，例如按钮和文本框，就要复杂一些。

既然控件是调用函数的结果，那它怎能从用户那里得到任何信息呢？答案实际上很聪明：显示控件的函数也会**返回**信息给它们的调用者。

同样，解释这一点最好的方法是提供一个示例。

3. Button

首先使用立即模式 GUI 系统创建一个按钮。

(1) **使用下面的代码替换 `OnGUI` 方法：**

```
private void OnGUI() {
  using (var verticalArea
    = new EditorGUILayout.VerticalScope()) {
      var buttonClicked = GUILayout.Button("Click me!");
      if (buttonClicked) {
       Debug.Log("The custom window's " +
         "button was clicked!");
      }
  }
}
```

调用 `GUILayout.Button` 方法时，会发生两件事：屏幕上将出现一个按钮；另外，如果在此区域刚完成鼠标单击，此方法将返回 `true`。

这个系统之所以能够工作，是因为 `OnGUI` 被重复调用。当窗口第一次出现时，调用 `Button` 导致屏幕上出现一个按钮。当用户把鼠标移动到该按钮上并按下鼠标按键时，`OnGUI` 再次被调用，GUI 系统将按钮绘制为“按下”状态。当用户松开鼠标按键时，`OnGUI` 再次被调用。因为单击已经完成，所以这一次调用 `Button` 会返回 `true`。

实际上，可以这样看待这类编程风格：`GUILayout.Button` 在屏幕上同步绘制一个按钮，并且如果用户单击了该按钮，就返回 `true`。

(2) **返回 Unity**，注意现在出现了一个按钮。单击该按钮时，Console 选项卡中将显示“The custom window’s button was clicked!”。



的确，是有点奇怪。但是，能怎么办呢？

4. 文本字段

按钮是允许用户提供信息的最简单的控件类型：用户要么单击了按钮，要么没有单击。不过，GUI 系统还支持更复杂的控件类型。例如，文本字段能够完成两个任务：向用户显示一些文本，**并**允许用户编辑该文本。

用来显示文本字段的方法是 `EditorGUILayout.TextField`。调用此方法时，需要提供一个字符串，即文本字段中显示的文本。然后，该方法返回用户在此文本字段中**输入**的内容，可能与你提供的内容不同。

为了使此方法能够工作，用来存储文本的变量不能是局部变量。也就是说，下面的代码是**不能**正确工作的：

```
private void OnGUI() {
  using (var verticalArea
    = new EditorGUILayout.VerticalScope()) {
```

```
            string textValue = "";

            textValue
              = EditorGUILayout.TextField(textValue);
  }
}
```



TextField 包含在 EditorGUILayout 类中，而不是 GUILayout 类中。GUILayout 也包含一个 TextField 方法，但是功能与 EditorGUILayout 类中的 TextField 方法不同。

如果在 Unity 中测试这段代码，可以在文本字段中输入内容，但是离开该文本字段后，其内容将重置为空字符串。

正确做法是，用来存储文本的变量必须是此类的一个变量：

```
private string stringValue;
private void OnGUI() {
  using (var verticalArea
    = new EditorGUILayout.VerticalScope()) {

    this.stringValue
      = EditorGUILayout.TextField(this.stringValue);
  }
}
```

在对 OnGUI 的不同调用之间，stringValue 的内容被保存，因此这段代码能够工作。

TextField 控件显示单行文本。如果想显示多行文本，可以使用 TextArea：

```
this.stringValue = EditorGUILayout.TextArea(
  this.stringValue,
  GUILayout.Height(80)
);
```



因为这两个控件使用了同一个变量，所以会显示相同的文本。另外，对一个控件进行修改时，另一个控件的内容会随之自动修改。这个效果很酷。

在上面这个例子中，通过提供 GUILayout 选项，可以覆盖文本区域的高度。此选项可添加到**任意**控件，如果需要一个比较高的按钮，可以向任意按钮添加对 GUILayout.Height(80) 的调用，该按钮的高度将变为 80 像素。

延迟文本字段　延迟文本字段是另外一种类型的文本字段。这种文本字段的工作方式与普通文本字段类似，只不过它们返回的值会保持你一开始提供的值，直到它们丢失焦点（即用户移动到另外一个文本字段，或者单击其他地方）才会改变。

当需要验证用户输入的数据时，这种字段很有用。不过在用户表明自己已完成输入之前，进行验证可能没有意义。

使用 DelayedTextField 方法可创建一个延迟文本字段，如下所示：

```
this.stringValue
  = EditorGUILayout.DelayedTextField(this.stringValue);
```

特殊文本字段　除了处理普通文本以外，文本字段也可以用来处理数字。特别是 TextField 控件，有 4 个非常有用的变体：整数字段、浮点数字段、Vector2D 字段和 Vector3D 字段。

例如，假设类中已经具有如下字段：

```
private int intValue;

private float floatValue;

private Vector2 vector2DValue;

private Vector3 vector3DValue;
```

我们可以创建字段，为上述字段提供数据：

```
this.intValue
  = EditorGUILayout.IntField("Int", this.intValue);

this.floatValue
  = EditorGUILayout.FloatField("Float", this.floatValue);

this.vector2DValue
  = EditorGUILayout.Vector2Field("Vector 2D",
                                 this.vector2DValue);
this.vector3DValue
  = EditorGUILayout.Vector3Field("Vector 3D",
                                 this.vector3DValue);
```



注意，第一个参数使用的是字符串：如果提供此参数，那么文本字段前面将显示一个标签。

5. 滑动条

文本字段除了用来获取数字输入，还可以提供一个图形滑动条。例如，可以像下面这样使用 IntSlider：

```
var minIntValue = 0;
var maxIntValue = 10;
this.intValue
  = EditorGUILayout.IntSlider(this.intValue,
                              minIntValue,
                              maxIntValue);
```

当结合 IntField 或 FloatField 控件，并且与这些控件使用相同的变量时，滑动条特别有用，因为可以移动滑动条来快速设置值。不过，如果需要设置一个非常具体的值，可以直接键入。

还可以使用最小值－最大值滑动条来规定最小值和最大值。例如，假设有两个类变量用来存储最小值和最大值：

```
private float minFloatValue;
private float maxFloatValue;
```

使用 MinMaxSlider 方法可绘制一个最小值－最大值滑动条：

```
var minLimit = 0;
var maxLimit = 10;
EditorGUILayout.MinMaxSlider(ref minFloatValue,
                             ref maxFloatValue,
                             minLimit,
                             maxLimit);
```

注意，此方法并不返回值，而是会修改传入的 minFloatValue 和 maxFloatValue 变量。另外，minLimit 和 maxLimit 值限制了 minFloatValue 和 maxFloatValue 可被设置的最小值和最大值。

6. 空白

空白（Space）控件是完全不可见的，其作用只是向 UI 添加空白。空白控件可用于在视觉上将控件拆分为不同的分组：

```
EditorGUILayout.Space();
```

7. 列表

目前为止，我们讨论的允许用户输入的控件都是相当宽松的：用户可以输入任意文本或数字。但是有些时候，我们想让用户从一个预定义选项的列表中做出选择。

为了支持这种功能，可以使用一个 Popup 列表。Popup 使用一个字符串选项数组，以及当前选定的数组项的一个整数。当用户改变当前选择的数组项时，当前选定项的数字也会改变。

例如，在类中添加下面的变量：

```
private int selectedSizeIndex = 0;
```

然后在 OnGUI 方法中添加下面的代码：

```
var sizes = new string[] {"small","medium","large"};

selectedSizeIndex
  = EditorGUILayout.Popup(selectedSizeIndex, sizes);
```

但是，记住 selectedSizeIndex 中存储的数字代表什么值可能很麻烦。更好的方法是使用**枚举**（enumeration 或者 enum）。

之所以说枚举更好，是因为编译器会检查它们。在上面的例子中，你需要记住 0 的含义是 small，但是直接说 Small 会更方便。枚举就能够实现这种功能。

下面定义一个枚举来表示几种不同的伤害类型。我们还将添加一个变量，用来存储当前选中的伤害类型。

(1) 将下面的代码添加到 TextureCounter 类中：

```
private enum DamageType {
  Fire,
  Frost,
  Electric,
  Shadow
}

private DamageType damageType;
```

通过使用此枚举和 damageType 变量，我们可以创建一个 Popup 列表来显示这个列表中的值。

(2) **将下面的代码添加到 OnGUI 方法中：**

```
damageType
  = (DamageType)EditorGUILayout.EnumPopup(damageType);
```

这将显示一个 Popup，其中包含了 DamageType 枚举能够代表的全部值，并且该 Popup 被设为当前选中的 damageType 变量的值。



需要将其强制转换为正确的枚举类型，因为 EnumPopup 方法并不知道自己使用的枚举类型。

8. 滚动视图

如果你一直在添加本章介绍过的不同控件，可能会注意到，控件开始超出编辑器窗口的边界。为了解决这个问题，可以使用**滚动视图**来让用户能够滚动界面。

滚动视图需要跟踪滚动位置。因此，需要创建一个变量来存储滚动位置，就像对其他控件所做的处理一样。

(1) **在 TextureCounter 类中添加下面的变量：**

```
private Vector2 scrollPosition;
```

创建滚动视图的方式与创建垂直列表十分相似：在 using 语句内创建一个新的 EditorGUILayout.ScrollViewScope。

(2) **在 OnGUI 方法中添加下面的代码：**

```
using (var scrollView =
  new EditorGUILayout.ScrollViewScope(this.scrollPosition)) {
  this.scrollPosition = scrollView.scrollPosition;

  GUILayout.Label("These");
  GUILayout.Label("Labels");
  GUILayout.Label("Will be shown");
  GUILayout.Label("On top of each other");
}
```

(3) **返回 Unity**，标签将包含在一个可滚动的区域中。可能需要调整窗口的大小，才能看到滚动效果。

16.2.2 AssetDatabase

在结束对编辑器窗口的讨论之前，我们回归 TextureCounter 窗口的目标：使其统计项目中的纹理数，然后显示到一个标签中。

为此，我们将使用 AssetDatabase 类。此类是访问项目中当前包含的所有资源的入口，可获得 Unity 控制的所有文件的信息，以及对这些文件进行修改。



我们没有足够的篇幅来讨论 AssetDatabase 类的全部功能；因此，强烈建议读者查阅 Unity 手册的 AssetDatabase 页面（https://docs.unity3d.com/Manual/AssetDatabase.html）。

(1) **使用下面的代码替换 TextureCounter 中的 OnGUI 方法：**

```
private void OnGUI() {
  using (var vertical = new EditorGUILayout.VerticalScope()) {
    //获取所有纹理的列表
    var paths = AssetDatabase.FindAssets("t:texture");

    //获取计数
    var count = paths.Length;

    //显示标签
    EditorGUILayout.LabelField("Texture Count",
      count.ToString());

  }

}
```

(2) **返回 Unity，在项目中添加一些图片**。什么图片并不重要，目的只是拖入一些文件。如果不知道添加什么，那就去搜索猫咪的图片。

现在，编辑器窗口将显示你添加的纹理数量。

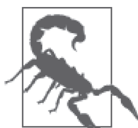
16.3 创建自定义属性绘制器

除了创建完全自定义的编辑器窗口，还可以扩展 Inspector 窗口的行为。

Inspector 的作用是提供一个用户界面，用来配置当前选中的游戏对象所附加的每个组件。对于每个组件，Inspector 会显示一个控件，代表该组件的每个变量。

对于常用类型，如字符串、整型和浮点型，Inspector 已经知道如何提供合适的控件。但是，如果你定义了一个自定义类型，Inspector 就不一定知道如何提供合适的控件了。一般来说这不是问题，但是界面有可能变得混乱。

这时，**属性绘制器**就可以一展身手。我们可以向 Unity 提供代码，决定如何把不同类型的数据提供给用户。



GUI 布局系统在自定义属性绘制器内不起作用，因此需要手动布局控件。不必担心，实际上没有听起来那么困难。我们将在示例代码中进行演示。

为了演示这一点，我们将创建一个自定义类，代表一系列值。这个类可供任何脚本使用。然后，我们将为这个自定义类定义一个自定义属性绘制器。为此，执行下面的步骤。

(1) **创建一个新的 C# 脚本**，命名为 Range.cs，保存到 Assets 文件夹中。

(2) **在 Range.cs 中添加下面的代码**：

```
[System.Serializable]
public class Range  {

  public float minLimit = 0;
  public float maxLimit = 10;

  public float min;
  public float max;

}
```



`System.Serializable` 特性将此类标记为能够保存到磁盘。这也告诉 Unity，类的值应该显示在 Inspector 中。

(3) **再创建一个 C# 类**，命名为 RangeTest，也保存到 Assets 文件夹中。这是使用 `Range` 类的一个简单的脚本组件。在 RangeTest.cs 中添加下面的代码：

```
public class RangeTest : MonoBehaviour {

  public Range range;

}
```

(4) **创建一个空游戏对象**，然后把 `RangeTest` 脚本拖放到这个对象上。

选择该游戏对象后，Inspector 将显示原始值（如图 16-7 所示）。

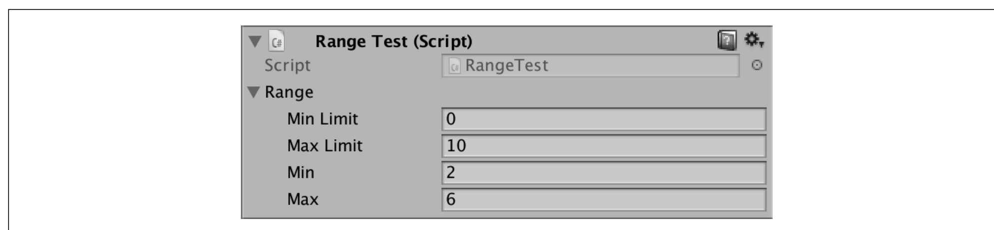


图 16-7：显示 `Range` 类默认界面的 Inspector

为了覆盖这个界面，我们将实现一个新类，替换 Unity 提供的默认界面。

(5) **创建一个新脚本**，命名为 RangeEditor.cs，保存到 Editor 文件夹中。

(6) **用下面的代码替换 RangeEditor.cs 的内容：**

```
using UnityEngine;
using System.Collections;

using UnityEditor;

[CustomPropertyDrawer(typeof(Range))]
public class RangeEditor : PropertyDrawer {

  //此属性绘制器将有两行，一行用于滑动条，
  //另一行用于文本字段，可在其中直接修改值
  const int LINE_COUNT = 2;

  public override float GetPropertyHeight (
    SerializedProperty property, GUIContent label)
  {
    //返回此属性高度的像素数量
    return base.GetPropertyHeight (property, label)
      * LINE_COUNT;
  }

  public override void OnGUI (Rect position,
    SerializedProperty property, GUIContent label)
  {

    //获取代表此Range属性内字段的对象
    var minProperty = property.FindPropertyRelative("min");
    var maxProperty = property.FindPropertyRelative("max");

    var minLimitProperty
      = property.FindPropertyRelative("minLimit");
    var maxLimitProperty
      = property.FindPropertyRelative("maxLimit");

    //PropertyScope内的任何控件都能正确处理预设——在预设中
    //修改的值将加粗，并且可以右击某个值，选择将其重置为预设的值
    using (var propertyScope
      = new EditorGUI.PropertyScope(
        position, label, property)) {

            //显示标签；此方法返回其周围的内容可以包含的一个矩形
            Rect sliderRect
              = EditorGUI.PrefixLabel(position, label);

            //为每个控件构造矩形：

            //计算单行的高度
            var lineHeight = position.height / LINE_COUNT;

            //滑动条高度是单行行高
            sliderRect.height = lineHeight;
```

```
            //两个字段的区域与滑动条具有相同的形状，但是下移了一行
            var valuesRect = sliderRect;
            valuesRect.y += sliderRect.height;

            //计算出两个文本字段的矩形
            var minValueRect = valuesRect;
            minValueRect.width /= 2.0f;

            var maxValueRect = valuesRect;
            maxValueRect.width /= 2.0f;
            maxValueRect.x += minValueRect.width;

            // 取出浮点值
            var minValue = minProperty.floatValue;
            var maxValue = maxProperty.floatValue;

            //进行修改判断，以正确地支持多对象编辑
            EditorGUI.BeginChangeCheck();

            //显示滑动条
            EditorGUI.MinMaxSlider(
              sliderRect,
              ref minValue,
              ref maxValue,
              minLimitProperty.floatValue,
              maxLimitProperty.floatValue
            );

            //显示字段
            minValue
              = EditorGUI.FloatField(minValueRect, minValue);
            maxValue
              = EditorGUI.FloatField(maxValueRect, maxValue);

            //有值发生变化了吗?
            var valueWasChanged = EditorGUI.EndChangeCheck();
            if (valueWasChanged) {
              //存储修改后的值
              minProperty.floatValue = minValue;
              maxProperty.floatValue = maxValue;
            }
    }

  }
}
```

这段代码量很大，所以我们将分块介绍。

16.3.1 创建类

首先，我们需要定义 `RangeEditor` 类，并告诉 Unity，在为 Inspector 遇到的任何 `Range` 属性绘制界面时使用这个类。这是使用 `CustomPropertyDrawer` 特性实现的，该特性接受 `Range` 类的类型作为参数。

另外，将 RangeEditor 的超类设为 PropertyDrawer。

```
[CustomPropertyDrawer(typeof(Range))]
public class RangeEditor : PropertyDrawer {
```

16.3.2　设置属性的高度

属性在 Inspector 中会占据一定的纵向空间。默认情况下，高度为 20 像素左右。但是，Range 属性需要更多空间，因为我们想绘制 Range 的滑动条，并在其下方绘制两个文本字段。

GetPropertyHeight 方法负责返回属性的高度，单位为像素。通过覆盖这个方法可改变属性的高度。

不要硬编码具体的数值，因为在不同的 Unity 版本中，数值可能发生变化。相反，我们将想要获得的行数定义为一个常量 LINE_COUNT，然后调用 base 实现来获得单行的高度，再将其乘以 LINE_COUNT。

```
//此属性绘制器将有两行，一个用于滑动条，
//另一个用于文本字段，可在其中直接修改值
const int LINE_COUNT = 2;

public override float GetPropertyHeight (
  SerializedProperty property, GUIContent label)
{
  //返回此属性高度的像素数量
  return base.GetPropertyHeight (
    property, label) * LINE_COUNT;
}
```

16.3.3　覆盖OnGUI

现在就可以实现这个类中的主方法：OnGUI。对于属性绘制器而言，这个方法有以下 3 个参数。

- position 参数是一个 Rect，定义了 OnGUI 方法绘制其控件的位置，以及可用区域的面积。
- property 参数是一个 SerializedProperty 对象，用来与这个特定类实例提供的组件的 Range 属性交互。
- label 参数是一个 GUIContent 对象，代表应该作为此属性的标签显示的一些图形内容，通常是一些文本。

```
public override void OnGUI (Rect position,
  SerializedProperty property, GUIContent label)
{
```

16.3.4　获取属性

属性绘制器的工作是呈现及修改组件内的一个属性。你并不会直接修改组件自身；相反，property 参数作为中介来协调访问。这意味着 Unity 能够提供附加的功能，例如支持自动撤消操作。

对于 Range 对象而言，这个 property 参数包含**其他**的属性。min、max、minLimit 和 maxLimit 变量本身都是属性，所以我们需要访问它们：

```
//获取代表此Range属性内的字段的对象
var minProperty = property.FindPropertyRelative("min");
var maxProperty = property.FindPropertyRelative("max");

var minLimitProperty
  = property.FindPropertyRelative("minLimit");
var maxLimitProperty
  = property.FindPropertyRelative("maxLimit");
```

16.3.5 创建属性作用域

除了获取代表属性的对象，我们还需要告诉 GUI 系统，绘制的控件关联到具体的属性。

这么做意味着 Unity 能够在需要时自定义控件的外观。例如，当包含属性的对象是某个预设修改后的实例时，让属性加粗显示。另外，当右键单击修改后的属性时，Unity 将打开一个菜单，允许将其值重置为预设。

为了支持上述功能，需要把所有控件放到一个 `Property Scope` 中：

```
using (var propertyScope
  = new EditorGUI.PropertyScope(position, label, property)) {
```

16.3.6 绘制标签

我们现在使用 `PrefixLabel` 控件绘制标签。此控件在 `position` 矩形内绘制 `label` 文本，然后返回一个新的 `Rect`，代表控件能够在标签旁边绘制的**剩余区域**。

这么做意味着属性的布局将遵循 Unity 的其余部分建立的风格：属性的标签位于左上角，字段位于右侧；标签下方的区域留空。

```
Rect sliderRect = EditorGUI.PrefixLabel(position, label);
```

16.3.7 计算矩形

知道了能够绘制控件的可用空间的大小后，我们需要计算 3 个控件（滑动条和两个文本字段）的矩形大小。

首先计算单行的高度，即将**总**空间除以 `LINE_COUNT`，单位为像素。然后，将 `sliderRect` 的高度设为新计算出的 `lineHeight`，同时保留其宽度。这意味着滑动条将占据整个顶行。

然后计算两个文本字段的矩形大小。这两个文本字段将并排显示在滑动条下方。为了计算它们的矩形，我们计算出代表整个第二行的矩形，然后将其分成两半：

```
var lineHeight = position.height / LINE_COUNT;

//滑动条高度是单行行高
sliderRect.height = lineHeight;

//两个字段的区域与滑动条具有相同的形状，但是下移了一行
var valuesRect = sliderRect;
valuesRect.y += sliderRect.height;

//计算出两个文本字段的矩形
```

```
var minValueRect = valuesRect;
minValueRect.width /= 2.0f;

var maxValueRect = valuesRect;
maxValueRect.width /= 2.0f;
maxValueRect.x += minValueRect.width;
```

16.3.8　获取值

因为 `MinMaxSlider` 直接修改传入的变量，所以我们需要将 `minProperty` 和 `maxProperty` 的值临时存入变量。最终，当我们即将绘制的控件修改了这些值以后，会把它们重新保存到属性对象中：

```
var minValue = minProperty.floatValue;
var maxValue = maxProperty.floatValue;
```

16.3.9　设置检查修改

在介绍绘制控件的核心内容之前，还需要做最后一项设置。我们需要让 Unity 告诉我们，即将绘制的任何控件的值是否发生了变化。

这个步骤很重要，因为如果不这么做的话，那么每次绘制控件时，我们都要修改属性，即使做出的修改未被应用。

这通常没有问题，但是如果选择了**多个**对象，并且它们都有 Range，那么显示 Range 控件的操作也将把它们**全部**改为一个值，即使用户什么也没有做。检查修改可以避免这种意外。

```
EditorGUI.BeginChangeCheck();
```

16.3.10　绘制滑动条

我们终于可以开始绘制控件了。我们已经有了控件需要显示的数据，存储返回结果的方法，以及用来包含控件的矩形。

首先绘制 `MinMaxSlider`：

```
EditorGUI.MinMaxSlider(
  sliderRect,
  ref minValue,
  ref maxValue,
  minLimitProperty.floatValue,
  maxLimitProperty.floatValue
);
```

16.3.11　绘制字段

接下来绘制文本字段。注意，我们使用的变量就是传入 `MinMaxSlider` 的变量。这意味着修改滑动条也将更新文本字段，反之亦然：

```
minValue = EditorGUI.FloatField(minValueRect, minValue);
maxValue = EditorGUI.FloatField(maxValueRect, maxValue);
```

16.3.12　检查修改

最后，我们可以询问 Unity，自从我们开始检查修改之后，是否有任何控件发生了变化。如果回答是肯定的，那么 `EditorGUI.EndChangeCheck` 方法将返回 `true`：

```
var valueWasChanged = EditorGUI.EndChangeCheck();
```

16.3.13　存储属性

如果修改了控件，我们就需要把新值存储到属性中：

```
if (valueWasChanged) {
  //存储修改后的值
  minProperty.floatValue = minValue;
  maxProperty.floatValue = maxValue;
}
```

16.3.14　进行测试

现在，我们就完成了全部设置工作。

返回 Unity，并查看 Inspector，可以看到 `Range` 变量的一个自定义 UI（如图 16-8 所示）。

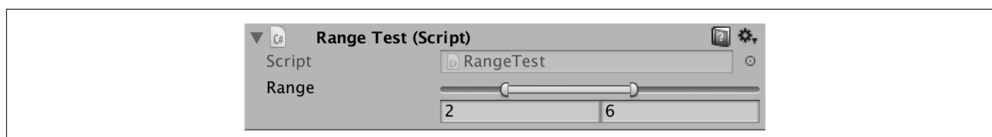


图 16-8：自定义的属性绘制器



可能需要先取消选中游戏对象，再重新选中，才能使用户界面更新。

编写了这些代码后，**任何**脚本上的**任意** `Range` 属性都将获得此自定义界面。

16.4　创建自定义Inspector

本章最后将讨论如何创建完全自定义的 Inspector。除了自定义单个属性的外观，还可以替换组件在 Inspector 内的整个用户界面。

为了说明如何实现这种功能，我们将首先创建一个简单的组件，然后为该组件创建一个全新的 Inspector 界面。

16.4.1　创建一个简单脚本

这个简单的组件将在游戏启动时，修改网格的颜色。

(1) 创建一个新脚本，命名为 RuntimeColorChanger。
(2) 将 RuntimeColorChanger 类更新为如下代码：

```
public class RuntimeColorChanger : MonoBehaviour {

  public Color color = Color.white;

  void Awake() {
    GetComponent<Renderer>().material.color = color;
  }
}
```

(3) 返回 Unity。打开 GameObject 菜单，选择 3D Object → Capsule。
(4) 将 RuntimeColorChanger 脚本拖放到该对象上。
(5) 将 RuntimeColorChanger 的 Color 属性改为红色，然后单击 Play 按钮。胶囊体将变为红色。

16.4.2 自定义Inspector的创建

目前为止还不错，脚本准确实现了我们想要的功能。

现在，我们想创建一个自定义 Inspector，来添加一个很酷的功能：一个按钮列表，能够快速将颜色修改为某个预定义颜色。我们将创建自定义 Inspector 来添加这些按钮。

(1) 创建一个脚本，命名为 RuntimeColorChangerEditor.cs，保存到 Editor 文件夹中。
(2) 用下面的代码替换 RuntimeColorChangerEditor.cs 的内容：

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic; //Dictionary需要
using UnityEditor;

//这是用于RuntimeColorChanger的编辑器
[CustomEditor(typeof(RuntimeColorChanger))]
//可以处理同时编辑多个对象的情况
[CanEditMultipleObjects]
class RuntimeColorChangerEditor : Editor {

  //一个string-color对的集合
  private Dictionary<string, Color> colorPresets;

  //代表所有选中对象的color属性
  private SerializedProperty colorProperty;

    //当编辑器第一次显示时调用
    public void OnEnable() {

        //设置颜色预设值列表
        colorPresets = new Dictionary<string, Color>();

        colorPresets["Red"] = Color.red;
        colorPresets["Green"] = Color.green;
        colorPresets["Blue"] = Color.blue;
        colorPresets["Yellow"] = Color.yellow;
```

```
        colorPresets["White"] = Color.white;

        //获取当前选中的对象的属性
        colorProperty
          = serializedObject.FindProperty("color");
    }

    //调用此方法在Inspector中绘制GUI
    public override void OnInspectorGUI ()
    {
        //确保serializedObject最新
        serializedObject.Update();

        //开始一个垂直的控件列表
        using (var area
          = new EditorGUILayout.VerticalScope()) {

          //对于预设值列表中的每个颜色……
          foreach (var preset in colorPresets) {

            //显示一个按钮
            var clicked = GUILayout.Button(preset.Key);

            //如果单击了按钮，就更新属性
            if (clicked) {
              colorProperty.colorValue = preset.Value;
            }
          }

          //最后，显示一个字段，允许直接设置颜色值
          EditorGUILayout.PropertyField(colorProperty);
        }

        //应用任何发生了变化的属性
        serializedObject.ApplyModifiedProperties();
    }
}
```

同样，我们将详细解释这段代码。

16.4.3 设置类

第一步是定义类及其在 Unity 系统中的角色。我们将 RuntimeColorChangerEditor 类定义为 Editor 类的子类。

另外，我们为其添加了 CustomerEditor 特性，指出应该将这个类作为任意 RuntimeColorChanger 组件的编辑器。最后，给该类添加 CanEditMultipleObjects 特性（顾名思义，该特性可同时编辑多个对象）：

```
//这是用于RuntimeColorChanger的编辑器
[CustomEditor(typeof(RuntimeColorChanger))]
//可以处理同时编辑多个对象的情况
[CanEditMultipleObjects]
class RuntimeColorChangerEditor : Editor {
```

16.4.4 定义颜色和属性

此类需要存储两条主要信息。首先，我们需要一个预定义颜色列表，供用户从中选择。另外，我们需要一个对象来代表当前选中的所有对象的 `color` 属性。

与创建自定义属性绘制器时类似，我们使用 `SerializedProperty` 对象代表属性。这意味着 Unity 能够为我们提供额外的功能，例如撤消：

```
//一个string-color对的集合
private Dictionary<string, Color> colorPresets;

//代表所有选中对象的color属性
private SerializedProperty colorProperty;
```

16.4.5 设置变量

当选中一个包含 `RuntimeColorChanger` 组件的对象时，Inspector 将为其创建一个编辑器。然后调用 `OnEnable` 方法，这是能够做一些设置的第一个机会。在此编辑器中，我们准备好 `colorPresets` 字典，在该字典中填入预定义颜色。

另外，我们需要获得待处理的 `color` 属性。为此，我们访问 `serializedObject` 变量，该变量由 Unity 设置，代表当前选中的所有对象。

```
public void OnEnable() {

    //设置颜色预设值列表
    colorPresets = new Dictionary<string, Color>();

    colorPresets["Red"] = Color.red;
    colorPresets["Green"] = Color.green;
    colorPresets["Blue"] = Color.blue;
    colorPresets["Yellow"] = Color.yellow;
    colorPresets["White"] = Color.white;

    //获取当前选中对象的属性
    colorProperty = serializedObject.FindProperty("color");
}
```

16.4.6 开始绘制GUI

在 `OnInspectorGUI` 中，我们可以实现自定义 Inspector。第一步是让 `serializedObject` 把自己更新到游戏场景的当前环境中，这确保了我们即将绘制的控件将准确代表场景：

```
public override void OnInspectorGUI ()
{
    //确保serializedObject最新
    serializedObject.Update();
```

16.4.7 绘制控件

现在可以绘制这个组件的控件了。我们使用 `VerticalScope`，为 `colorPresets` 字典中的每

个预设值绘制一个按钮。如果单击其中任意一个按钮，那么其 colorProperty 值将设为对应预设值的颜色值。

绘制按钮后，我们为颜色显示一个 PropertyField。PropertyField 控件显示一个适合属性类型的控件。在本例中，因为 colorProperty 代表 RuntimeColorChanger 中的 color 变量，所以将显示一个颜色选择器，允许用户选择自己的颜色。这样，我们就能让用户为对象做出精细的选择，同时能够提供额外的功能：

```
using (var area = new EditorGUILayout.VerticalScope()) {

  //对于预设值列表中的每个颜色……
  foreach (var preset in colorPresets) {

    //显示一个按钮
    var clicked = GUILayout.Button(preset.Key);

    //如果单击了按钮，就更新属性
    if (clicked) {
      colorProperty.colorValue = preset.Value;
    }
  }

  //最后，显示一个字段，允许直接设置颜色值
  EditorGUILayout.PropertyField(colorProperty);
}
```

16.4.8　应用修改

最后，让选中的对象（可以是多个对象）应用修改。这是通过调用 serializedObject 的 ApplyModifiedProperties 方法实现的：

```
//应用任何发生了变化的属性
serializedObject.ApplyModifiedProperties();
```

16.4.9　进行测试

现在就可以测试自定义 Inspector。

选择游戏对象，将看到自定义 Inspector（如图 16-9 所示）。可能需要先取消选中胶囊体，然后再次选中。

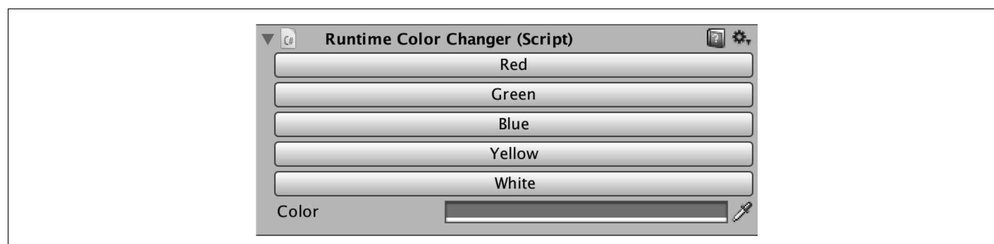


图 16-9：自定义 Inspector



显示默认的 Inspector 内容

有时候并不需要**替换**某个组件的 Inspector，只想在 Inspector 中添加一些额外的内容。此时，可以使用 `DrawDefaultInspector` 方法来快速绘制 Inspector 通常包含的内容，然后可以在这些内容的上方或下方绘制额外的控件：

```
public override void OnInspectorGUI() {

    //绘制默认的Inspector控件
    DrawDefaultInspector();

    //向开发人员显示一条鼓励消息
    var msg = "You're doing a great job! " +
      "Keep it up!";

    EditorGUILayout.HelpBox(msg, MessageType.Info);
}
```

16.5 小结

自定义编辑器能够大大简化你的工作。如果需要执行重复性任务，或者需要一种更好的方法来查看对象中包含的数据，那么编辑器能够提供很大的帮助。但是要记住，玩家不会看到你的自定义编辑器。因为这些编辑器存在的意义是帮助开发人员，所以不要纠结于创建完美的自定义编辑器，它们能够帮助你创造什么才是真正重要的。

第 17 章

编辑器之外

游戏已经构建完成，你也完善了游戏玩法，看起来效果不错。接下来该做什么呢?

现在是时候走出 Unity 编辑器了。利用 Unity 提供的诸多有用的服务，可以改进自己的游戏，改进制作游戏的方式，甚至能够让游戏为你带来源源不断的收入。在本章中，我们将介绍这 3 种做法。

我们还将讨论为设备构建游戏，并使游戏走向更广阔的世界。

17.1　Unity服务生态系统

当人们讨论 Unity 的时候，通常指的是 Unity 编辑器，即 Unity Technologies 公司开发和销售的软件。但是，Unity 并不只是一个编辑器。除了软件之外，Unity 还提供了许多服务，可用来提高开发人员的生活质量。Asset Store、Unity Cloud Build 服务和 Unity Ads 平台是其中最重要的 3 个服务。

17.1.1　Asset Store

Unity Asset Store 是一个网上商店，程序员、艺术家以及其他游戏内容制作者可以在这里出售能够被集成到游戏中的内容。

Asset Store 特别适合那些缺少某种技能的人。例如，不懂（或者没有时间）制作美术资源的程序员能够购买他们需要的 3D 模型，从而只关注自己更擅长的编程。同理，需要音频、游戏脚本等资源的人也可以在这里购买。Asset Store 的内容巨细无遗——在这个商店中，你可以购买一辆汽车的 3D 模型，也可以购买适合特定游戏类型的完整资源包。



从 Asset Store 购买的资源——特别是**优秀**的资源——很容易被认出是从这个商店中购买的。需要注意的是，过度依赖 Asset Store 资源会让你的游戏显得单调。

Asset Store 提供的资源中，有一些值得特别注意，因为它们为 Unity 添加了一些本身不具备的功能。

1. PlayMaker

PlayerMaker 是一个可视化脚本工具，由 Hutong Games 创建。在可视化脚本系统中，通过连接预定义的代码模块，来定义游戏对象的行为，这些预定义代码模块表示为方框，框中有线条延伸出来。

可视化脚本系统是除编写代码之外的另一种方案，对于编程新手来说往往也更加容易上手。它们特别适合表示严重依赖状态的行为。例如，敌人 AI 随机游动，直到看到玩家，此时敌人 AI 进入**追寻**状态，并袭击玩家，直到自己死亡、玩家死亡，或者看不到玩家。

PlayMaker 可在 Asset Store 上购买（https://assetstore.unity.com/packages/tools/visual-scripting/playmaker-368）。

安装 PlayMaker　因为 PlayMaker 为定义游戏行为提供了完全不同的方法，所以有必要仔细进行探讨，并设置一些简单的行为。在执行下面的步骤之前，需要先从 Asset Store 购买 PlayMaker，在撰写本书时（2017 年年中），其价格为 65 美元。



我们将在一个针对 3D 图形配置的新的空项目中执行下面的步骤。

(1) **下载并安装软件包**。安装窗口将会显示（如图 17-1 所示）。



图 17-1：PlayMaker 的安装窗口，在导入安装包后显示

(2) **单击 Install**。PlayMaker 将检查你的项目，确保能够安装，并且确保你的软件版本是最新的。



如果你没有使用版本控制工具（如 Git），PlayMaker 会给出警告。你可以忽略这个警告，但是总体来说，使用版本控制工具是一个好主意。

(3) 在第二个安装窗口（如图 17-2 所示）中，**单击 Install 按钮**，然后在弹出的对话框中单击“I Made a Backup, Go Ahead!”，Unity 将导入另外一个包。
(4) 在显示的窗口中，**单击 Import 按钮**。



图 17-2：第二个安装窗口



依你的 Unity 版本而定，安装过程中可能询问你，Unity 是否可以把代码更新到与最新 API 兼容。必须选择同意，然后才能继续操作。

安装过程完成后，关闭任何仍然打开的窗口。现在就可以开始使用这个工具了。

使用 PlayMaker　PlayMaker 的构造是基于**有限状态机**（Finite State Machines，FSM）的概念。有限状态机是一个逻辑系统，在这种系统中，一个对象只能是多种**状态**中的一种，每种状态都能够改变或**过渡**到这些状态的一个预定义子集。也就是说，如果有“坐着”“站着”和“跑动”这几种状态，那么从“站着”可以过渡到“坐着”或者“跑动”，但是不能直接从“坐着”过渡到“跑动”。当状态改变时，就能够运行某种行为。

在这个简单的教程中，添加的行为也极为简单：我们将创建一个球，当球落到一个表面上时，就会改变颜色。

首先来设置好环境。

(1) 打开 GameObject 菜单，选择 3D Object → Sphere，**创建球体**。
选择新创建的对象，然后使用 Inspector 中的 Transform 组件，将其位置设为 (0, 15, 0)。
(2) **为球体添加一个 Rigidbody 组件**。
(3) 再次打开 GameObject 菜单，选择 3D Object → Plane，**创建地面**。
将此对象的位置设为 (0, 0, 0)。
(4) 最后，**将 Camera 的位置设为** (0, 9, −16)，旋转设为 0。这将使摄像机同时看到球体和地面。

场景现在应该如图 17-3 所示。

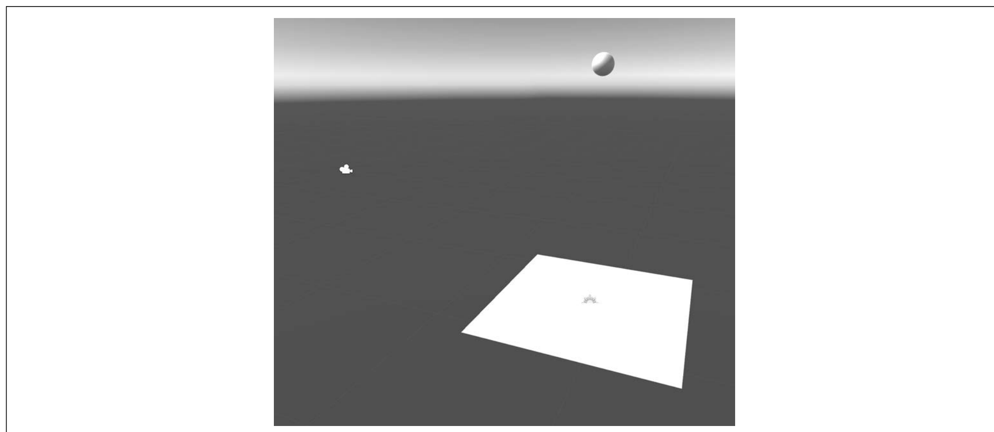


图 17-3：为本教程布置好的场景

现在，我们开始为球体添加 PlayMaker 行为。

(1) **打开 PlayMaker 编辑器**。打开 PlayMaker 菜单，然后选择 PlayMaker Editor。
PlayMaker Editor 选项卡将会显示（如图 17-4 所示）。



你可能觉得把该选项卡附加到 Unity 窗口会更方便。为此，只需将该选项卡从窗口顶部拖放到你喜欢的位置即可。



图 17-4：PlayMaker 编辑器

(2) **为球体添加一个 FSM**。选择 Sphere，然后在 PlayMaker 窗口中右键单击并选择 Add FSM。



PlayMaker 窗口会显示许多提示，它们很有用，但是会占据不少空间。要禁用提示，可以按 F1 键，或者单击 PlayMaker 窗口右下角的 Hints 按钮。

默认情况下，FSM 将包含一个状态，名为 `State1`。在我们的演示程序中，有两个状态：`Falling` 和 `HitGround`。我们将重命名第一个状态，然后添加另一个状态。

(3) **将第一个状态重命名为** `Falling`。选择 `State1` 状态，进入 PlayMaker 窗口右侧的 State 选项卡，然后将其名称改为 `Falling`。

(4) 在 PlayMaker 窗口中右键单击并选择 Add State，**添加** `HitGround` **状态**。将新状态重命名为 `HitGround`。

FSM 现在应该如图 17-5 所示。



图 17-5：添加状态后的 FSM

当球撞击地面时，我们想让状态发生改变。为此，我们将创建从 `Falling` 状态到 `HitGround` 状态的**过渡**，当 FSM 附加到的对象与其他对象发生碰撞时就会触发过渡。

(5) 右键单击 `Falling` 状态，选择 Add Transition → System Events → COLLISION ENTER，**添加过渡**。这将显示一个新过渡，带有一个警告，指出还没有把此过渡连接到一个目标状态（如图 17-6 所示）。

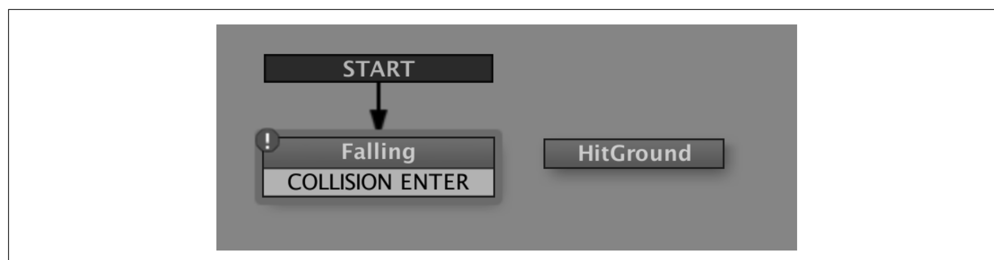


图 17-6：添加了过渡，但是还没有连接过渡时的 FSM

(6) **将过渡连接到** `HitGround` **状态**。左键单击 COLLISION ENTER 过渡，将其拖放到 `HitGround` 状态上。此时将显示一个箭头，将二者连接起来（如图 17-7 所示）。

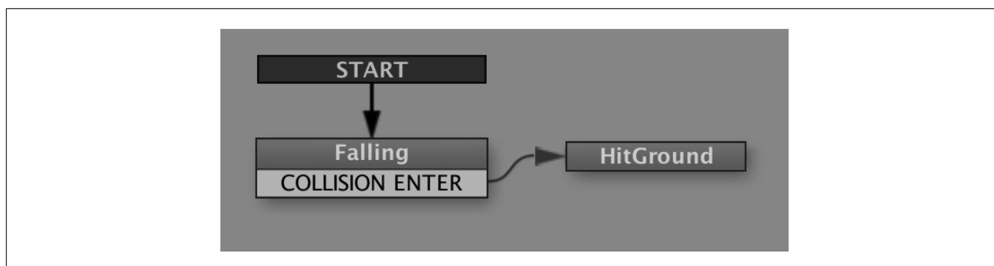


图 17-7：连接状态后的 FSM

(7) 按 Play 按钮**测试游戏**。FSM 窗口将高亮显示当前的状态。`Falling` 动作将一直高亮显示，直到球体碰到地面。

接下来要添加当对象进入 `HitGround` 状态时运行的动作。具体来说，我们想让材质改变颜色。

(1) **为 `HitGround` 状态添加 Set Material Color 动作**。选择 `HitGround` 状态，进入 State 选项卡，然后单击 Action Browser。Action Browser 窗口将会显示。向下滚动，找到并单击 Material 按钮，然后选择 Set Material Color 项（如图 17-8 所示）。单击 Add Action to State 按钮，动作将显示在 State 选项卡中（如图 17-9 所示）。

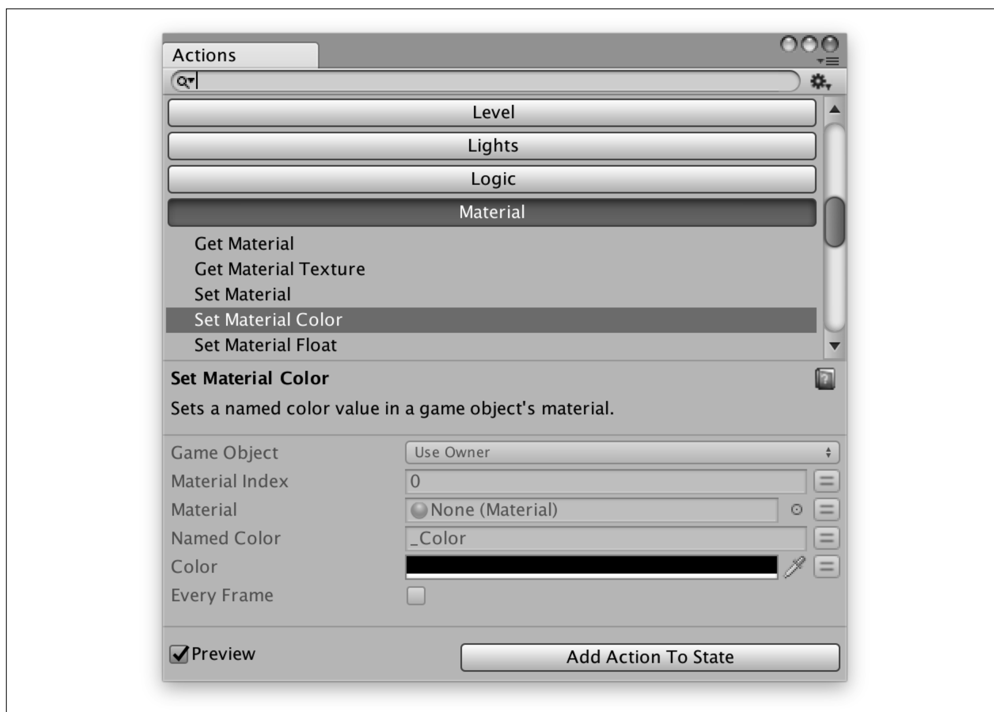


图 17-8：Action Browser

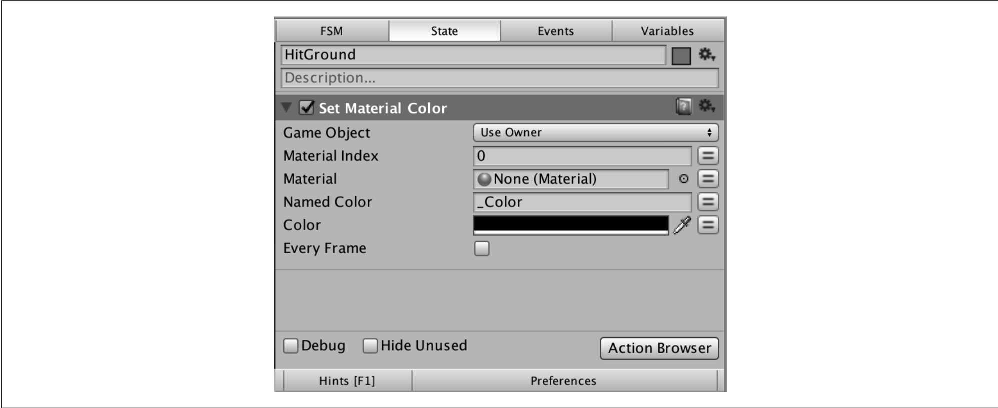


图 17-9：添加并配置动作后的 FSM

(2) **使颜色变为绿色**。在 State 选项卡中，将颜色改为绿色。

(3) **测试游戏**。当球体触碰到地面时，将改变为绿色。

2. Amplify Shader Editor

第 14 章介绍过，着色器一般需要编写代码。但是，相比游戏玩法的代码，着色器的视觉本质使它们更适合可视化构造——相比于编写代码来将两个代表颜色的向量相乘，实际**看到**这个过程会更加直观。

Amplify Shader Editor（如图 17-10 所示）就是为 Unity 设计的几种可视化着色器编辑器之一。通过把节点连接起来，Amplify Shader Editor 能够创建和演示你的材质，并生成可在游戏中使用的资源。这通常比自己编写着色器代码更快、更容易，对认为以可视化方式创建视觉结果更加直观的人特别有用。

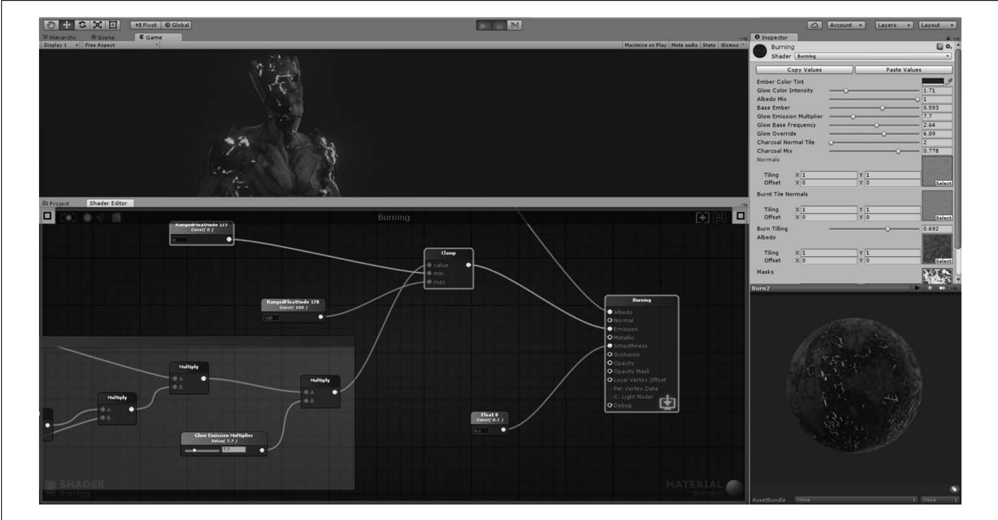


图 17-10：Amplify Shader Editor

Amplify Shader Editor 可在 Asset Store 上购买（https://assetstore.unity.com/packages/tools/visual-scripting/amplify-shader-editor-68570）。

3. UFPS

UFPS，也叫 Ultimate FPS（如图 17-11 所示），是第一人称射击游戏的简单基础。虽然 Unity 也提供了第一人称控制器，但是没有包含第一人称游戏常用的其他功能，例如弯腰、爬梯子或者与按钮交互。UFPS 提供了这些功能的实现，还提供了专门用于射击类游戏的功能，例如管理武器库、管理弹药以及管理玩家生命值。



虽然 UFPS 主要面向构建动作类射击游戏，但是也同样适合慢节奏的游戏。Fullbright 的游戏 *Gone Home*（2014）就使用 UFPS 来处理第一人称表现，这个游戏的玩法就是在屋子内四处走动，查看遗留的物体、文件和家具。



图 17-11：Ultimate FPS

UFPS 可在 Asset Store 上购买（https://assetstore.unity.com/packages/templates/systems/ufps-ultimate-fps-2943）。

17.1.2 Unity Cloud Build

为目标平台构建项目是一个复杂的过程，需要强大的处理能力和大量时间。虽然你**可以**在自己的计算机上构建项目，但是这么做并不总是最**合适**的做法，尤其是当项目很大、很复杂时。

Unity Cloud Build 服务下载你的源代码并进行构建，然后供你下载构建后的版本（如果构建失败，会通知你）。当配置 Could Build 关注你的源代码存储库时，它将观察源代码存储库的变化。当源代码改变时，Unity 将自动构建你的游戏。



你当然可以创建自己的构建服务器，而不使用 Cloud Build。但是，这样做的过程很烦琐，而且会占用你的许可包含的两次激活之一。Cloud Build 减少了你的控制权，但是为你提供了易用性。

在撰写本书时，Cloud Build 是一项免费服务。如果你拥有 Unity Plus 订阅，则你的构建将被优先处理，能够更快完成。如果你拥有 Unity Pro 订阅，那么你的构建能够并发运行；因此，如果你的游戏被设计为在多个平台上运行（例如 iOS 和 Android），那么两种构建将同时开始。

Unity 的网站上提供了关于 Cloud Build 的更多信息（https://unity3d.com/cn/unity/features/cloud-build）。

17.1.3　Unity Ads

Unity Ads 服务能够在你的游戏中显示全屏视频广告。当玩家观看广告时，你将得到一笔小费用。通过这种方法，你能够为自己的游戏开发一种额外的收入来源。

奖励式广告是视频广告的一种特例：作为一种交换，玩家观看完广告后，能够得到某种游戏内奖励，例如游戏币奖励、改变皮肤或其他内容。

如何让游戏为自己带来收益是一个庞大的主题，需要参阅专门的图书。想要使用 Unity Ads，可以查看 Unity 网站的服务页面（https://unity3d.com/cn/unity/features/ads）。

17.2　部署

当你准备好让游戏走出编辑器，进入实际的设备时，需要使用 Unity **构建**游戏。这涉及 3 个步骤：把游戏的全部资源捆绑到一起，编译游戏脚本，把构建好的应用安装到设备上。Unity 将为你完成前两个步骤，但是第三个步骤需要你自己完成。

本节将介绍如何为 iOS 和 Android 设备构建游戏。动手之前，我们需要先简单介绍应该执行的设置工作，以及 Unity 不同版本之间的区别。

17.2.1　设置项目

你可以在任何时候构建自己的项目。但是为了获得最好的结果，应确保游戏的玩家设置是正确的。玩家设置包括游戏的名称和图标，以及其他的一些设置，这些设置可能控制着运行游戏的屏幕方向，以及向安装游戏的操作系统标识游戏的唯一 ID 字符串等。

为了进行配置，需要访问 Player Settings。打开 Edit 菜单，选择 Project Settings → Player。Inspector 将如图 17-12 所示。

图 17-12：PlayerSettings Inspector



有些设置在多个平台上是相同的。例如，游戏的名称和图标在不同平台之间不大可能发生改变。Unity 在这种设置的名称旁边加上一个星号（*），说明它们是多个平台通用的设置。

每个应用程序（无论是运行在 iOS 还是 Android 上）都需要以下设置。

- 游戏的**产品名称**，显示在主界面和市场中。
- 游戏**制作公司的名称**，用于市场中。
- 游戏的**图标**，用于主界面和市场中。
- 游戏的**闪屏**，在游戏启动时显示。

- 游戏的**唯一标识符**（bundle identifier），这是一条文本，在市场上唯一标识游戏，并且不会显示给用户。唯一标识符的构建方法是，将你拥有的域名（例如 oreilly.com）颠倒过来，然后再加上游戏的名称（例如 com.oreilly.MyAwesomeGame）。

测试游戏需要配置游戏名称和标识符。要将游戏发布到 iTunes App Store 或者 Google Play 商店中，需要具有上述所有元素。

默认情况下，产品名被设为项目的名称，公司名被设为 DefaultCompany。如果对默认设置感到满意，可以保留产品名称不变（如图 17-12 的顶部所示）。

要改变唯一标识符，从 Cursor Hotspot 下的菜单中选择要针对哪个平台构建，然后打开 Other Settings 节。在这里，将 Bundle Identifier 设为你想要使用的标识符（如图 17-13 所示）。

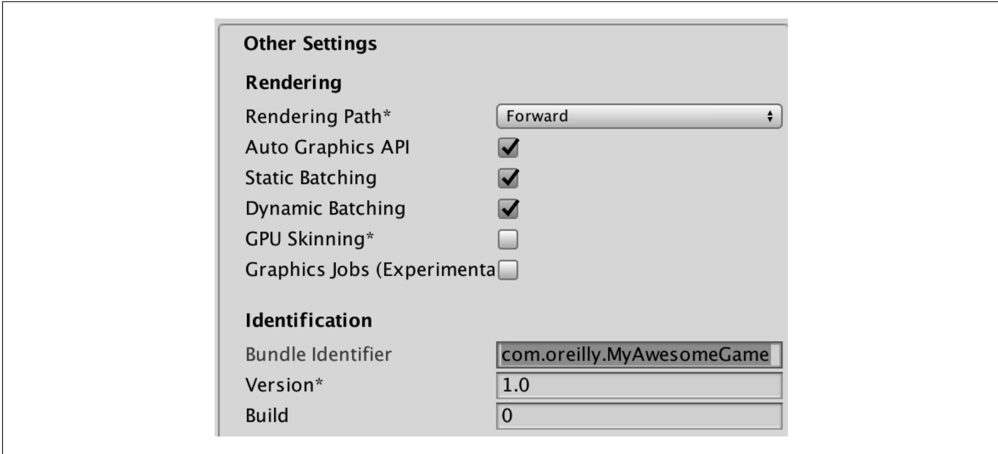


图 17-13：设置项目的唯一标识符



如果同时为 iOS 和 Android 构建游戏，不需要将唯一标识符设置两次。此设置在所有平台之间是共享的，这也适用于游戏的版本号和其他几种设置。

17.2.2 设置目标

你一次只能选择一个目标。默认情况下，Unity 将目标设为 PC、Mac 以及 Linux Standalone，将具体目标设为运行 Unity 的机器。例如，如果你使用的是 Mac，那么默认目标是 macOS；如果你使用的是 PC，那么默认目标是 Windows。



下载平台模块

为了针对特定平台而构建，Unity 首先需要针对该平台安装正确的模块。首次安装 Unity 时，安装程序会询问你想为哪个平台安装模块。如果没有平台相应的安装模块，Build Settings 窗口将如图 17-14 所示。你需要单击该窗口中的按钮，下载并安装合适的模块。

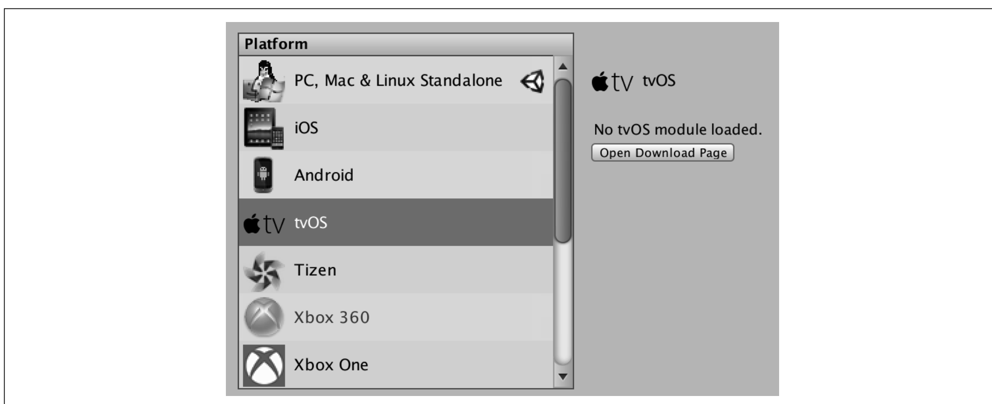


图 17-14：Build Settings 窗口，选中平台的模块未被加载

因为在本书第二部分和第三部分中，设计和构建的是移动游戏，所以首先要做的是切换到期望的平台。打开 File 菜单，选择 Build Settings 选项，这会打开 Build Settings 窗口（如图 17-15 所示）。



图 17-15：Build Settings 菜单

在此窗口中，只需要选择目标平台，然后单击窗口左下角的 Switch Platform 按钮。



切换平台时，Unity 将重新导入游戏的全部资源。如果项目很大，这个过程将需要很长时间。一定要耐心等待。为了改善体验，Unity 提供了一个 Cache Server 工具来保存导入资源的副本，更多信息请查阅文档（https://docs.unity3d.com/Manual/CacheServer.html）。

闪屏

需要指出的是，免费版与加强版和专业版构建的应用是有区别的。免费版用户在启动自己的游戏时必须显示一个闪屏，加强版和专业版用户则可以选择禁用这个闪屏。

这个闪屏相当克制，它显示 Unity 标志，以及文本 Made with Unity，并且只显示两秒钟，同时游戏的初始场景在后台加载。



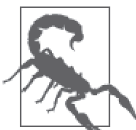
无论使用哪个 Unity 版本，都可以在很大程度上自定义闪屏。除了显示 Unity 标志，还可以包含你自己的标志、自定义背景颜色、设置背景图片和不透明度，以及将闪屏设置为同时或者按次序显示多个标志。自定义闪屏可打开 Edit 菜单，选择 Project Settings → Player，然后向下滚动到 Splash Image → Splash Screen。Unity 为此主题提供了大量文档，要了解更多信息，请查阅相关文档（https://docs.unity3d.com/Manual/class-PlayerSettingsSplashScreen.html）。

17.2.3　针对平台构建游戏

为 iOS 构建游戏和为 Android 构建游戏的步骤并不相同，下面将分别介绍。

1. 针对iOS构建游戏

Unity 使得构建自己游戏的 iOS 版本十分简单。本节中将详细介绍如何让自己的游戏运行在 iPhone 上。



目前，要针对 iOS 构建游戏，只能使用 macOS 计算机，或者通过 Unity Cloud 构建（此服务在 Mac 上进行构建）。

把游戏直接部署到自己的个人设备上是完全免费的。把游戏分发给其他人，需要通过 iTunes App Store。这意味着需要注册 Apple Developer Program，费用为一年 99 美元，注册网址为 https://developer.apple.com/programs/。

首先需要下载 Xcode，也就是 iOS 的开发环境。

(1) **下载 Xcode**。启动 Mac App Store，搜索 Xcode，然后下载。
(2) 完成下载后，**启动 Xcode**。

接下来需要配置 Xcode 来使用自己的账户。无论你是否注册了付费的 Apple Developer Program，Xcode 都需要使用你的 Apple ID，将你注册为开发人员，这样在设备上安装游戏

之前才能进行必要的代码签名。

(1) **打开 Xcode 菜单，选择 Preferences**。在窗口顶部，单击 Accounts 按钮。单击左下角的 Add 按钮（+），从弹出的菜单中选择 Add Apple ID。

(2) **将设备连接到计算机的 USB 端口**。

现在就配置好了 Xcode，可以开始构建 Unity 游戏了。

(1) **返回 Unity，打开 Build Settings 窗口**。打开 File 菜单，选择 Build Settings 按钮，这将打开 Build Settings 窗口。

(2) **选择 iOS 平台，然后单击 Switch Platform 按钮**。Unity 将把项目切换到 iOS（如图 17-16 所示）。这可能需要几分钟的时间。

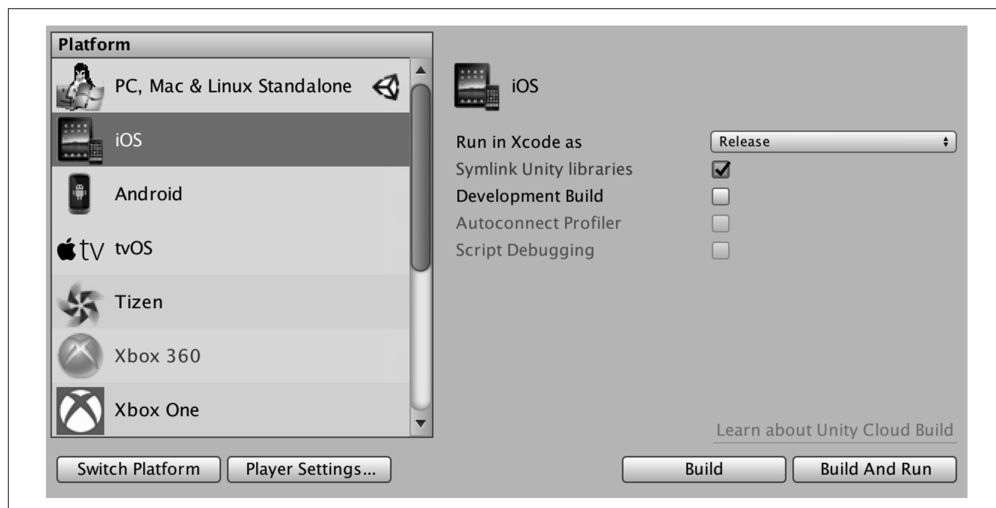


图 17-16：使用 iOS 平台的 Build Settings 窗口



为了节约空间，可打开 Symlink Unity Libraries 按钮。这样就不会把整个 Unity 库（几百 MB 大小）复制到项目中。

(3) **单击 Build and Run 按钮**。Unity 会询问将项目保存到什么地方。选择一个文件夹后，Unity 将为 iOS 构建应用，然后在 Xcode 中打开该项目，并告诉 Xcode 在连接的设备上构建并运行游戏。



代码签名问题

如果出现关于代码签名的错误，可选择窗口左上角的项目，然后选择 Unity-iPhone 目标，从 Team 下拉菜单中选择开发团队（可能只是你自己的姓名），最后单击 Fix Issue 按钮（如图 17-17 所示）。这将整理你的证书，并解决问题。再次按 Command-R 来尝试构建。

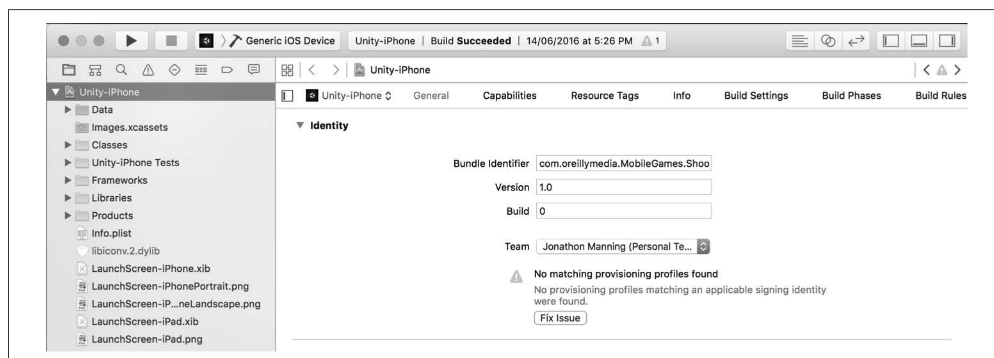


图 17-17：Xcode 中的 Fix Issue 按钮

2. 针对Android进行构建

要针对 Android 进行构建，首先需要安装 Android SDK。Android SDK 负责把构建好的应用部署到设备上。

(1) 从 Android Developer 网站**下载 Android SDK**：http://developer.android.com/sdk。

(2) 按照下面网页中的说明**安装 SDK**：http://developer.android.com/sdk/installing/index.html。



如果使用的是 Windows 操作系统，可能需要先下载一个 USB 驱动程序，才能让计算机与 Android 设备通信。下载地址是 http://developer.android.com/sdk/win-usb.html。如果使用的是 macOS 或 Linux，则不需要此驱动程序。

现在可以告诉 Unity，Android SDK 安装到了什么位置。

(1) **打开 Unity 菜单，选择 Preferences → External Tools**。在此窗口中，单击 SDK 字段旁边的 Browse 按钮，浏览并找到你安装 Android Studio 的文件夹。

(2) **打开 Build Settings 窗口**。打开 File 菜单，并选择 Build Settings 选项，这将打开 Build Settings 窗口。

(3) **选择 Android 平台，并单击 Switch Platform 按钮**。Unity 将把项目切换到 Android。

(4) **选择 Google Android Project**（如图 17-18 所示）。这么做意味着 Unity 将导出并生成一个在 Android Studio 中使用的项目。

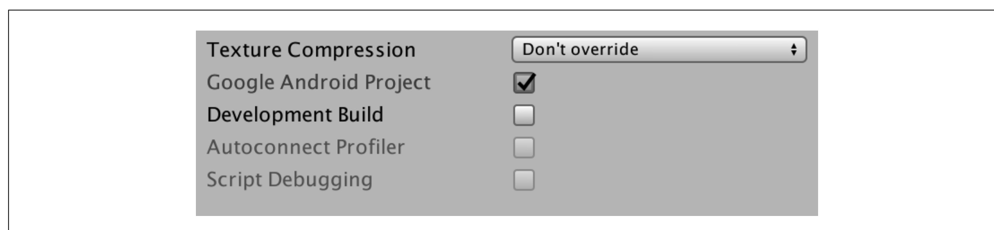


图 17-18：使 Android 构建生成一个 Google Android 项目

(5) **单击 Export 按钮**。Unity 将询问把项目保存到什么地方。选择位置后，将生成项目。

(6) **在 Android Studio 中打开项目，然后单击 Play 按钮**。项目将会编译，然后安装到你的手机上。

虽然在不同发布版本之间，Unity 的变化一般不会太大，但是设置和构建到移动设备的过程可能更加频繁地发生变化。考虑到这一点，我们不会在这里重复 Unity 的文档内容，因为到了本书出版时，可能这些内容已经过时了。相反，我们建议你阅读下面给出的 Unity 文档，这些文档很有用，用逐步介绍的教程形式说明了如何针对 Android 和 iOS 平台进行设置：

- “Getting started with Android development”（https://docs.unity3d.com/Manual/android-GettingStarted.html）
- “Getting started with iOS development”（https://docs.unity3d.com/Manual/iphone-GettingStarted.html）



iOS 和 Android 开发都涉及下载和安装大量软件。建议你在一个网速不错的地方进行设置。

17.3　拓展资料

欢迎来到这里：本书的结束部分。如果你一路读到这里，那么你已经完成了一段很长的旅途，从头开始构建了两个完整的游戏，并将 Unity 控制在手中，能够根据自己的需要来自定义 Unity。



如果你直接翻到了本书结尾，那么告诉你，结尾就是这个样子。你给自己剧透了。

在说再见之前，我们在此列出了一些有用的资源，供你参考并拓展自己的技能。

- Unity 的文档非常好，可作为整个编辑器的参考手册。该文档分成两个部分：描述编辑器的手册（https://docs.unity3d.com/Manual/index.html），以及描述 Unity 脚本的每个类、方法和函数的脚本编写参考（https://docs.unity3d.com/ScriptReference/index.html）。将 Unity 文档作为参考资料十分方便。
- Unity 的官方论坛（https://forum.unity.com/）是社区讨论中心，在这里你能够得到需要的帮助。
- Unity Answers（https://answers.unity.com/index.html）是一个官方支持的问答论坛。如果你有具体的问题，先查阅这里是一个好主意。
- Unity 经常举行在线培训课程（https://unity3d.com/cn/learn/live-training），让他们的某个培训讲师以在线课堂的形式演示某个功能或者一个完整的项目。即使没能看到实时的课程也没有关系，他们一般会录制下来，供以后观看。

- 最后，Unity 提供了许多教程（https://unity3d.com/cn/learn/tutorials），既有新手入门内容，也有进阶的、更加具体的指导。

我们希望你在阅读本书的过程中有所收获。如果你制作出来一个游戏，也许项目很小，也许你认为自己做得还不够好，但是都没有关系，我们会很乐意聆听你的想法。你随时可以给我们发邮件（unitybook@secretlab.com.au）。

作者简介

乔恩·曼宁（Jon Manning）博士和帕里斯·巴特菲尔德－艾迪生（Paris Buttfield-Addison）博士共同创立了 Secret Lab 工作室，致力于构建游戏和游戏开发工具。他们于近期构建了 iPad 游戏 *ABC Play School*，协助开发了独立游戏 *Night in the Woods*，并构建了澳洲航空公司针对儿童常旅客开发的 iPad 游戏 *Qantas Joey Playbox*。

他们在 Secret Lab 创建了 YarnSpinner 叙事类游戏框架，并为 O'Reilly Media 撰写图书。

乔恩和帕里斯之前是 Meebo（已被 Google 收购）的移动开发人员和产品经理，并且均拥有计算机专业博士学位。

关于封面

本书封面上的动物是巨棘鬼竹节虫和天牛（天牛科）。

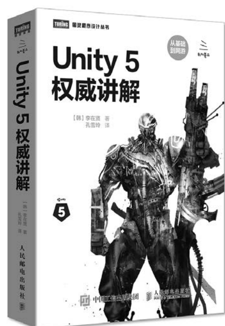
巨棘鬼竹节虫是澳大拉西亚本土的一种植食性无翅昆虫。雄性可长到 10~13cm 长，比较大的雌性一般为 15cm 左右。虽然大部分竹节虫生活在树上，但是巨棘鬼竹节虫生活在地面上（通常是雨林中），夜间搜寻食物，并通过伪装和假死来逃避捕食者。白天，它们成群聚集在脱落的树皮下和树洞中。这种昆虫是很受欢迎的宠物，雄性后肢上长长的尖刺（它们也因此得名）在巴布亚新几内亚被用作鱼钩。

天牛科甲虫拥有超长而强大的触角，常常可达到甚至超过其躯干的长度。天牛科包含 26 000 多种，从泰坦大天牛（世界上体形较大的昆虫之一，长度可达 32cm，不计入腿长），到极小的属 Decarthia（这个属只包含 3 个种，均只有几毫米长）。天牛科（Cerambycidae）的名称得自希腊神话人物 Cerambus，他原本是一个牧羊人，被一群仙女变成了一只甲虫。

O'Reilly 封面上的许多动物都濒临灭绝，它们都是这个世界的至宝。想要了解如何提供帮助，请访问 animals.oreilly.com。

封面插图由 Karen Montgomery 根据 J. G. Wood 的版画 *Insect Abroad* 绘制。

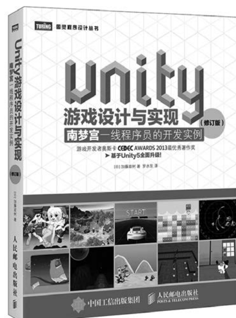
技术改变世界 · 阅读塑造人生



Unity 5 权威讲解

- 从基础到网游，专业开发人员讲述高效游戏制作技巧
- 以Unity 5为基础边做边学，直接实现各种游戏框架和功能
- 同时积累基础知识和实操技术，适合更多游戏界人士

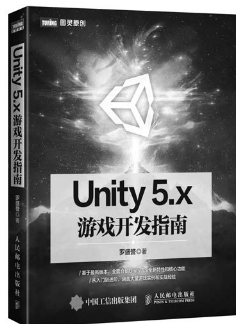
书号： 978-7-115-43636-8
定价： 109.00 元



Unity 游戏设计与实现（修订版）

- 游戏开发者奥斯卡CEDEC AWARDS 2013获奖图书，旧版豆瓣9.3分好评，基于Unity5全面升级
- 南梦宫主要开发者执笔，重点讲解设计思路和实现细节，公开灵感来源
- 详略得当，风格细腻，附带完整的工程源码

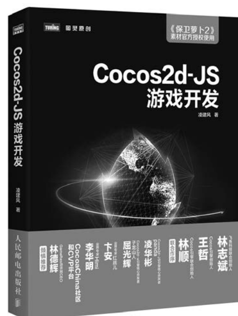
书号： 978-7-115-44899-6
定价： 79.00 元



Unity 5.x 游戏开发指南

- 全面介绍Unity 5.x全新特性和核心功能
- 从入门到精通，涵盖大量游戏实例和实战经验
- 一本让你迅速上手的游戏开发宝典

书号： 978-7-115-40364-3
定价： 69.00 元

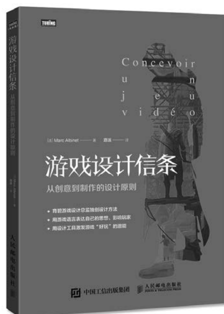


Cocos2d-JS 游戏开发

- 飞鱼科技联合创始人林志斌、Cocos引擎创始人王哲、Cocos引擎联合创始人林顺联合作序
- Cocos2d-JS引擎核心开发者panda（凌华彬）、Cocos2d-x引擎核心开发者子龙山人（屈光辉）、业界高手红孩儿（卞安）、业界高手Himi（李华明）、CocoaChina社区和CVP平台、GameRes游资网CEO林德辉倾情推荐
- 全面深入探讨基于JavaScript的游戏开发引擎Cocos2d-JS，无论新手还是老手都能获益良多

书号： 978-7-115-42148-7
定价： 69.00 元

技术改变世界 · 阅读塑造人生



游戏设计信条：从创意到制作的设计原则

- ◆ 《刺客信条》游戏设计总监、BAFTA游戏设计大奖获得者、育碧AAA级游戏设计师Marc Albinet独创设计秘辛
- ◆ 独具一格的剧本、场景与关卡设计方式，简单实用的工具和理念
- ◆ 从策划、创意、艺术创作到团队创意的工作技巧，实例丰富
- ◆ 游戏爱好者、游戏设计初学者与专业人士的资料上选

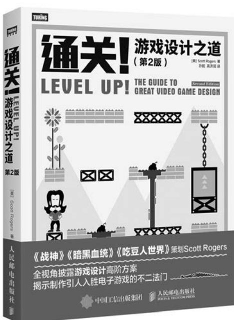
书号： 978-7-115-48021-7
定价： 49.00 元



大师谈游戏设计：创意与节奏

- ◆ 万代南梦宫主要制作人、《忍者龙剑传》《皇牌空战3》设计师吉泽秀雄执笔，从业30余年经验总结
- ◆ 详述寻找创意的思路，揭示让游戏好玩的秘诀，披露人气游戏创作背后的故事

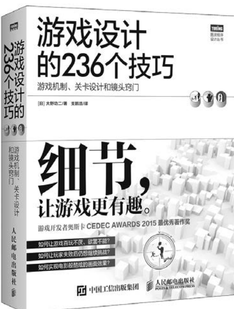
书号： 978-7-115-45669-4
定价： 39.00 元



通关！游戏设计之道（第 2 版）

- ◆ 《战神》《暗黑血统》《吃豆人世界》策划Scott Rogers，全视角披露游戏设计高阶方案，揭示制作引人入胜电子游戏的不二法门
- ◆ 详述AAA游戏设计细节，阐明游戏制作精髓

书号： 978-7-115-43177-6
定价： 99.00 元



游戏设计的 236 个技巧

- ◆ 荣获游戏开发者奥斯卡CEDEC AWARDS 2015著作奖
- ◆ 图文并茂、讲解清晰，告诉读者如何构造可玩性强、画面精美、让玩家爱不释手的游戏

书号： 978-7-115-40608-8
定价： 99.00 元

O'REILLY®

Unity移动游戏开发

本书是一本实用性与趣味性兼具的移动游戏开发指南，针对没有任何游戏开发经验的读者，通过精彩的项目、丰富的图片和细致的讲解，展示了如何使用Unity游戏引擎，构建适用于iOS和Android设备的游戏。

- **探索Unity基础知识，了解架构游戏、图形、脚本、音效、物理和粒子系统**
- **使用2D图形和物理功能构建卷轴动作游戏**
- **创建具有射击炮弹和重生对象功能的3D空战模拟游戏，并管理3D模型的外观**
- **深入了解Unity的高级功能，例如预计算光照、着色、自定义编辑器和游戏部署**

乔恩·曼宁（Jon Manning）和**帕里斯·巴特菲尔德–艾迪生（Paris Buttfield-Addison）**是Secret Lab的联合创始人。Secret Lab是一个独立游戏开发工作室，位于澳大利亚的塔斯马尼亚。

“想在移动平台上构建游戏，就必须了解Unity。Unity是开发人员理想的游戏开发引擎之一。本书有料有趣，全面介绍了Unity游戏开发，能指导你在Unity中创造自己的游戏玩法。”

——**Adam Saltsman**
*Canabalt*和*Overland*创作者，
Finji游戏工作室创始人

“学习如何使用游戏引擎，一定要亲自动手构建项目。本书将引导你创建两个截然不同的游戏，让你获得使用Unity多种功能的宝贵经验。”

——**Alec Holowka**
*Night in the Woods*和*Aquaria*的
主要开发者，Infinite Ammo
游戏工作室

“这本书改变了我的生活。我现在能感受到内心的平静，而且我确信我的目光可以穿透时间。”

——**Liam Esler**
澳大利亚游戏开发者协会

封面设计：Karen Montgomery 张健

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议 计算机／游戏开发

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)



ISBN 978-7-115-48879-4
定价：89.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱:
ebook@turingbook.com。

在这可以找到我们:

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks